Automated Reasoning for Planning Railway Infrastructure

Doctoral Dissertation by

Bjørnar Luteberget

Submitted to the
Faculty of Mathematics and Natural Sciences at the University of Oslo
for the degree Philosophiae Doctor in Computer Science

Date of submission: 2019-05-28
Date of public defense: 2019-10-18

Reliable Systems
Department of Informatics
University of Oslo
Norway

Oslo, May 2019

Abstract

Construction of railways is a challenging and expensive task. Comprehensive planning is required to ensure that the finished system has a high transportation capacity while remaining safe, reliable and maintainable. Computer science has a long history in the domain of railway analysis. Firstly, creating time tables that make efficient use of a large network of railway lines is assisted by numerical optimization and stochastic simulations. Secondly, the signalling and control systems that ensure that trains do not collide or derail, are modelled mathematically and proven correct using formal methods and automated reasoning.

However, both of these important tasks rely on a model of the infrastructure describing tracks, platforms, signals, sensors, and more. The infrastructure needs to be correctly designed to allow a control system to ensure safety without creating bottlenecks in the network. The design process for infrastructure is characterized by tedious, repetitive verification of elaborate specifications, and complex interplay with other disciplines, such as tracks, power lines, and telecommunications. Because the work is so complicated in this sense, essential questions about capacity and safety can go unanswered until it is too late to change the construction plans.

To help increase quality and efficiency, we have developed several components that contribute to the overall goal of bringing automated analysis tools to railway infrastructure engineering work. The main contributions are:

1. A system for **static analysis** of railway infrastructure based on Datalog logic.
2. A language of **local capacity specifications**, which can be used to relate the network-scale requirements to the construction of railway stations.
3. A **solver** which verifies local capacity specifications on proposed railway infrastructure based on Boolean satisfiability solvers and discrete event simulation.
4. An algorithm for **synthesis and optimization** of signal and sensor placements from a model of tracks and platforms, based on the verification for local capacity.
5. A **controlled natural language** for specifications so that engineers can access and maintain formalized rules and regulations in an understandable format.
6. A **visualization algorithm** for railway infrastructure, allowing automatic transfer of data from geographic or purely topological information to schematic views.
7. Integration of these tools with engineering software through the **railML** standard format and though graphical user interfaces built into **drafting software**.

The tools and techniques have been tested and evaluated using real-world railway infrastructure from the Norwegian railway network and from ongoing construction projects.

Contents

Abstract **iii**

Preface **vii**
Publications vii
Acknowledgements ix

1 Railway construction planning **1**
1.1 Research goals 1
1.2 Introduction to railway engineering 3
1.3 Software tool support for railway infrastructure design 9
1.4 Automated reasoning and formal methods 11
1.5 Research contributions 14

2 Static analysis in infrastructure verification **17**
2.1 Verification artifacts 18
2.2 Logic programming and knowledge-base systems 20
2.3 Tool implementation 33
2.4 Incremental verification 35
2.5 Conclusions 40

3 Dynamic analysis using local capacity specifications **43**
3.1 Capacity in railway construction projects 43
3.2 Dynamic behaviour 47
3.3 Performance requirement properties 52
3.4 Tool chain and solver architecture 54
3.5 Timing evaluation using simulation 66
3.6 Case studies and performance 72
3.7 Related work 74
3.8 Conclusions and future work 75

4 Optimization and synthesis of signalling component layouts **77**
4.1 Specification-based design synthesis 77
4.2 Design principles for local railway capacity 78
4.3 Algorithm for synthesis and optimization 82
4.4 Local optimizations and interactive improvement 88
4.5 Related work 89
4.6 Conclusions and future work 89

5 Controlled natural language specifications **91**
5.1 Participatory verification 92
5.2 Design methodology for a verification front-end language 97
5.3 RailCNL: a front-end language for railway verification 105
5.4 Evaluation of RailCNL coverage of Norwegian regulations 119
5.5 Tool integration 120
5.6 Conclusions 131
5.7 Excerpts from RailCNL Grammar as written in GF 134
5.8 Example content from regulations 135
5.9 Overview of Norwegian regulation contents 138

6 Drawing schematic plans **141**
6.1 Problem definition and formalization 143
6.2 Model definitions and drawing algorithms 147
6.3 Tool usage 158
6.4 Conclusions and Future Work 159

7 Integration with industrial end-user software **161**
7.1 Semantic CAD 162
7.2 Railway analysis tasks as CAD plug-in programs 165
7.3 Other external analysis tools 169

8 Conclusions **171**
8.1 Generality, maintenance, and user-friendliness 171
8.2 Design automation, quality, and standards compliance for tools . . 172
8.3 Railway engineering in the future 174

A Junction manual **177**
A.1 Overview 178
A.2 Getting started 181
A.3 Infrastructure 183
A.4 Dispatch 186
A.5 Planning 189
A.6 Tool windows 191

Bibliography **199**

Preface

This Ph.D. thesis is a result of the *RailCons*[1] project, a collaboration between Railcomplete AS and the University of Oslo. The project has been funded by Railcomplete AS and the Research Council of Norway under the Industrial Ph.D. scheme.

The project originated in the railway signalling engineering company Anacon AS, where engineers were dissatisfied with the state of tool support for performing design and engineering of railway signalling and interlocking systems. The company Railcomplete AS was founded as an offshoot from Anacon AS with the goal of integrating domain-specific analysis tools for signalling and other railway disciplines with cross-discipline drafting tools, so that engineering projects could be executed more efficiently and yield higher quality. Railcomplete AS also launched the RailCons research project to bring in research expertise on mathematical modelling and automated verification from the Reliable Systems research group at the Department of Informatics at the University of Oslo.

In this project, we have investigated the use of formal methods from computer science to handle the variety and complexity of specifications and regulations that a railway engineering project must comply with. We have used logic-based methods and automated solvers and integrated them with drafting tools. This thesis can be considered the final report on the results of the RailCons project

For me personally, the Ph.D. period has been an exciting exploration of state-of-the-art problem solving techniques from the field of computer science, made even more interesting by the fact that we have been solving real-world problems that originated in engineering practice. I am grateful for having had the chance to collaborate with very capable researchers, programmers, and engineers, and hope that our joint work will make an impact on how railway engineering is practised in the future.

Publications

The contents of this thesis is based on research articles that are either published in international, peer-reviewed computer science conference proceedings and journals, or are under review in such venues. The relevant articles are:

- Static analysis of railway infrastructure models and interlocking specifications (Chapters 2 and 7):
  - Bjørnar Luteberget, Christian Johansen, and Martin Steffen: *Rule-based Consistency Checking of Railway Infrastructure Designs.* International Conference

[1] RailCons web page: `https://www.mn.uio.no/ifi/english/research/projects/railcons/`

on Integrated Formal Methods (iFM 2016), [109]. Received award for best paper of the conference.

  - Bjørnar Luteberget, Christian Johansen, Claus Feyling, and Martin Steffen: *Rule-Based Incremental Verification Tools Applied to Railway Designs and Regulations*. International Symposium on Formal Methods (FM 2016), [108].
  - Bjørnar Luteberget and Christian Johansen: *Efficient Verification of Railway Infrastructure Designs Against Standard Regulations*. Formal Methods in System Design, [107].

- Controlled natural language specifications for railway verification (Chapter 5):
  - Bjørnar Luteberget, John J. Camilleri, Christian Johansen, and Gerardo Schneider: *Participatory Verification of Railway Infrastructure by Representing Regulations in RailCNL*. International Conf. on Software Engineering and Formal Methods (SEFM 2017), [105].
  - Bjørnar Luteberget, John J. Camilleri, Christian Johansen, and Gerardo Schneider: *RailCNL: A Controlled Natural Language for Railway Design Verification Specifications*, under review for publication as a journal article.

- Verification, synthesis, and optimization of infrastructure using local capacity specifications (Chapters 3 and 4):
  - Bjørnar Luteberget, Koen Claessen, and Christian Johansen: *Design-Time Railway Capacity Verification using SAT modulo Discrete Event Simulation*. Formal Methods in Computer-Aided Design (FMCAD 2018), [106]. Received award for best paper of the conference.
  - Bjørnar Luteberget, Koen Claessen, Christian Johansen, and Martin Steffen: *Design-Time Railway Capacity Verification using SAT modulo Discrete Event Simulation*, under review for publication as a journal article.
  - Bjørnar Luteberget, Christian Johansen, and Martin Steffen: *Optimization and Synthesis of Railway Signalling Layout from Local Capacity Specifications*. To appear in International Symposium on Formal Methods (FM 2019).

- Drawing of schematic railway plans (Chapter 6):
  - Bjørnar Luteberget, Koen Claessen, and Christian Johansen: *Drawing Schematic Railway Maps using SAT and Optimization*. To appear in International Conference on Integrated Formal Methods (iFM 2019).

I have been involved as a main contributor on each one of these articles, and they were written in their entirety between 2015 and 2019 for the purpose of supporting this thesis.

Acknowledgements

Many thanks to Christian Johansen and Martin Steffen at the University of Oslo for helping keeping the Ph.D. friction coefficient low, and for providing guidance along a (mostly) fixed track.

Special thanks to Claus Feyling, Thomas Benjaminsen, and all the employees at Railcomplete AS for funding and organizing the project, and for offering railway expertise.

Thanks to Gerardo Schneider and John J. Camilleri at Chalmers University for close collaboration on natural language specifications during my research visit to Göteborg from Nov. 2016 to May 2017.

Thanks to Koen Claessen at Chalmers University for creative problem solving related to capacity verification and schematic plans.

— Bjørnar Luteberget, Oslo, May 2019.

Railway construction planning 1

Research into improved methods for railway construction have the potential to create large savings in public spending in the years to come. Computer software tools for track layout and power catenary lines planning are already well developed. However, signalling and interlocking have had little progress in design tooling. The complex nature of the designs means that we need thoroughly researched methods to be able to ensure the correctness of the design with respect to safety regulations, as well as to optimize and automate the design processes.

Signalling engineering, in the context of constructing of railway infrastructure, consists of setting up signals, train detectors, derailers, and related equipment, and building a control system called the interlocking which ensures that all train movements happen in a safe sequence.

Comprehensive regulations and processes have been put in place to ensure the safety of such systems. At the same time, the locations of signalling components on the railway tracks can have crucial impact on the capacity of the railway, i.e. its ability to handle intended operational scenarios in a timely manner. There are many details of the signalling layout design which can cause operational scenarios to become infeasible or slow. Some examples are signal and detector placement, correct allocation and freeing of resources, track lengths, train lengths, etc. Capacity-related decisions in signalling border closely to the fields of timetable planning and the implementation of interlocking systems, and although tool support for verification of interlockings and optimization of timetables has been thoroughly investigated and developed since the beginnings of computer science (for example, the maximum flow problem was originally formulated to estimate railway network capacity [72]), signalling layout design still lacks appropriate modelling and analysis.

Signalling engineers are looking for a more agile working environment, supported by software tools, which would facilitate new railway designs to reach the decision makers considerably faster, and allow quickly adopting changes dictated by problems encountered in the field during the construction and implementation phases.

1.1 Research goals

The goal of this thesis is to introduce new kinds software tools to aid the design process of railways, focusing especially on the signalling. By signalling we mean:

- Positioning of signal lamps and signs (physical or virtual), train detectors, and related equipment, in the track models.

- Interlocking of movable elements, signals and train detection, resulting in a safe control system for dispatching trains using the signalling components.
- Automatic speed control mechanisms, such as the ATC system in use in Norway and Sweden, or the ETCS international standard.

Most of the process described above has so far been based on the engineer's experience and gut feeling. It can be hard to keep all the constraints in one's head at the same time, and often the design may be redone several times, and in the end it may not be optimal. By automating parts of this process we hope to create a more objectively good design, which can be shown to fulfil its functional requirements and allow for more rapid safety certification. To achieve this, we have aimed for the following goals:

- **Goal A**: **Formal specifications**. Capture specifications for railway engineering, such as rules, regulations, and project requirements, in formal logic so that maintenance and handling of complexity can be assisted by automated tools.
- **Goal B**: **User-friendly reasoning**. Assist railway engineers who are not educated in computer science and logic in working with formal specifications and associated tools.
- **Goal C**: **Design automation**. Synthesize models, starting from empty or partial models, to produce complete designs which are optimized for safety and performance.

To achieve these goals, we have brought together three different fields of engineering and research: firstly, the field of railway engineering, especially the signalling sub-discipline, which is the application domain that we are targeting. Secondly, software development for railway engineering, which is concerned with data representations of railway designs, and graphical user interfaces integrated into drafting programs. Thirdly, the computer science sub-discipline called formal methods concerns itself with handling automated reasoning and verification tasks where both general and project-specific specifications are a separate concern from the reasoning and verification technique. The following sections in this chapter give a brief introduction to each of these three topics, before we return in Section 1.5 to an overview of the rest of this thesis and how the research goals have been fulfilled.

1.2 Introduction to railway engineering

Railway is a mode of transport for passengers and goods where cars with wheels run on a pair of rails, called the *track*. There are two physical characteristics of railways, as opposed to paved roads, which are defining for its use:

- **(P1) Low friction**

  Using steel wheels on steel rails has a friction coefficient which is on average only one eighth of the friction between a car's rubber tires and asphalt road. This means that less energy is spent on accelerating and keeping a constant velocity, but it also means that the maximum braking power is reduced correspondingly. The distance required to brake the train often exceeds the sight range of the train driver for speeds higher than around 20 km/h.

- **(P2) Fixed guidance path**

  The path that vehicles take on a railway is fixed by the location of the tracks, so there is no possibility to go off-road. Together with low friction, this means that trains on collision course can neither stop nor manoeuvre around each other to avoid impact, so organizing traffic with safety in mind is of the highest importance. It also means that railway infrastructure must be carefully planned to include side tracks with space for vehicles to overtake and cross each other.

These physical characteristics give the railways the advantage that trains can be run with high speed and energy efficiency, but it also means that constructing railways is costly. However, high investment costs also means that the infrastructure required for powering vehicles with electric energy can be cost-effective. Together with the need for centralized control to ensure safety, this makes railways among the safest forms of transport with low levels of CO_2 emissions.

The high level of centralized control imposed on the train traffic in a railway system is essential to the design philosophy and engineering practice of the railway domain. Exactly how the infrastructure will be used in operation must be carefully considered in advance, and all the supporting systems of the railway (signalling, automatic speed control, catenary power, etc.) need to coordinate their designs in detail to make a safety-certified working system and to avoid bottlenecks. A computer scientist would say that the system is *tightly coupled*, which is known to cause complex inter-dependencies and make it hard and time consuming to implement and to adjust when requirements change.

Making a railway system run optimally depends on complex interplay between infrastructure (tracks and fixed equipment), rolling stock (trains with dynamical characteristics), and operations (procedures ensuring safety and time tables ensuring efficiency). It is the infrastructure itself which is the domain of this thesis, but already when designing the infrastructure, the available rolling stock and the intended operations must be kept in mind to produce an optimal design.

1.2.1 Railway infrastructure

The railway infrastructure consists of the following main parts:

- **Track network**: switches (also called turnouts or points) are a mechanical part of the railway tracks that allows trains to be guided from one track to another at junctions. The switches are set into a position by a point machine, which can be operated manually or by the interlocking (see Figure 1.1).

  The track network of tracks and switches is organized into *main tracks*, where trains travel at high speed on a time table, and *siding tracks* (also called secondary tracks), where train cars are stored and trains are assembled (see Figure 1.2).

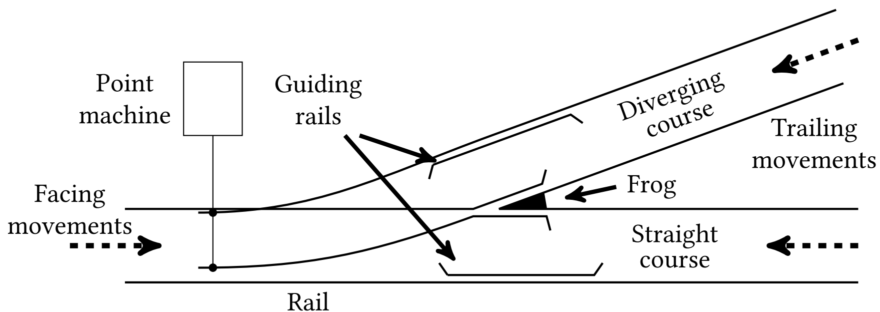


Figure 1.1: A railway switch allows trains to be guided from one track to another. Facing movements go into the straight or the diverging route, depending on the *setting* of the point machine.

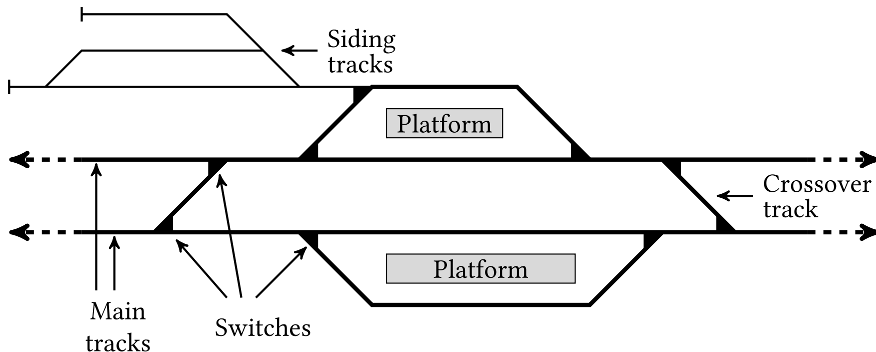


Figure 1.2: A railway network (here in *schematic* presentation) consists of tracks and switches. Tracks are categorized into main tracks (drawn in bold) and siding tracks. Crossovers are short tracks used for changing between two parallel tracks.

- **Signalling and interlocking**: in order to direct traffic and keep trains clear of each other, most railways use control systems called *interlockings*, which are designed to only allow movements which are safe. The interlocking uses *train detectors*, such as track circuits or axle counters, to determine whether a railway track is free of trains and safe for another train to move into. Whenever the interlocking allows a safe movement, it communicates this to the train using semaphores, signalling lamps, or radio communication. Signalling lamps are still most commonly used, and signals can communicate a fixed number of messages (called *aspects*) depending on how many lamps is has. *Main signals* indicate authority to move ahead from the current signal, while *distant signals* indicate whether the next main signal is already showing a proceed aspect, which allows the train to proceed at a higher speed. A main signal and a distant signal are often combined into one *combined* signal. The control system is designed according to the highest safety standards, but the safety relies also on the train driver correctly interpreting the signals and executing the movements. Most railways have also *automatic train control* which can override the train driver's control in case the train exceeds given limits. These systems rely on automatically transmitting data from the control system (infrastructure) to the train, typically using magnets, *balises*, or radio communication.

  Figure 1.3 shows examples of schematic symbolic representation of signalling components.

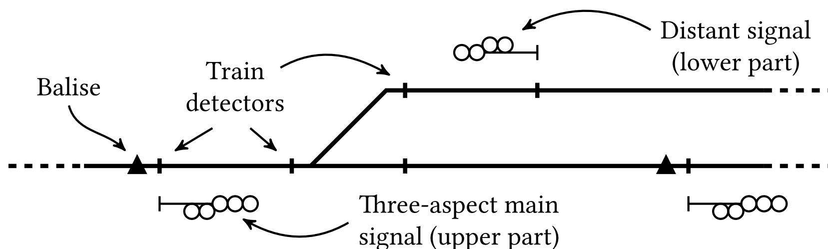


Figure 1.3: Railway signalling equipment used by the interlocking (control system) to ensure that a dispatcher can only request safe movements to be signalled to the train driver.

- **Telecommunications**: reliable digital and analogue transmission equipment and cables are necessary for communication between stations and control centres, between trains and train dispatchers, and between signalling equipment and the interlocking.

- **Catenary power supply**: many high-traffic railway lines have overhead electric wires that supply power to train engines. These wires carry high voltage elec-

tricity from feeder stations placed along the railway at regular intervals. Wires are broken into sections, each of which is suitably tensioned over a sequence of masts to avoid mechanical oscillations causing damage.

Designing railway infrastructure is *highly cross-disciplinary* because engineers in each of these sub-disciplines (track, signalling, telecommunications, catenary) perform their design under tight constraints, and are also dependent on each other's design decisions. For example, the location of the signals must be coordinated with the location of masts for the catenary power supply to avoid conflicts. The target maximum speed of the trains may be limited by the design of the catenary system, which again impacts the optimal location of signs and signals.

This thesis mainly concerns the design of the signalling and interlocking system and its relation to the track network, and also cross-disciplinary design constraints. Increasing the usage of analysis tools across disciplines and coordinating data models to use for this purpose increases the chances that conflicts between sub-disciplines can be caught early and corrected before the design is finished.

1.2.2 Control system: interlocking safety principles

The control system ensures safety by imposing the following conditions for allowing train movements:

- **(C1) Way integrity**

  The movable elements (switches and other mechanical parts of the track) along the path should be mechanically *locked* in place and *detected* to be in their correct positions to avoid derailing.

- **(C2) Path vacancy**

  The track should be *detected clear* of trains to avoid collisions.

- **(C3) Blocking conflicts**

  The track sections along the train's path should be *exclusively allocated* to the movement, and other movements should be performed at a safe distance.

The interlocking gets requests from a dispatcher to *set routes* and must then either ensure that conditions (C1), (C2) and (C3) are fulfilled before communicating movement authority to the train, or decline the request, see Figure 1.4.

The details of the workings of the interlocking vary greatly between countries. Many modern railways (especially those based on the German engineering tradition) describe the interlocking's workings using *tabular route-based specifications*. These specifications consist of a list of *elementary routes*, which describe requests to move a train from one main signal to an adjacent signal, together with the conditions for accepting the request.

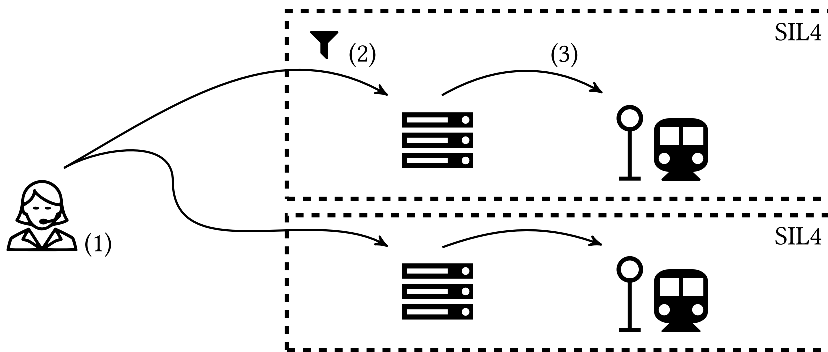


Figure 1.4: A dispatcher (1) requests routes from the interlocking control system. The interlocking decides whether to accept the command (2), and signals the resulting movement authority to the train driver (3). The control system itself is responsible for the safety of the resulting movements.

These conditions typically include:

- switch positions which must be mechanically locked,
- track sections which must be exclusively allocated, or which must be vacant,
- conflicting elementary routes: routes that use the same track sections, the same switches, or which for other reasons are not safe to use simultaneously, are often explicitly listed for each elementary route.
- flank protection: to protect the train from other movements, a certain distance along each path leading into the route path (the *flank*) should be detected vacant, and should have an element, e.g. a signal or a switch, locked in a protective state.
- safety zone (also called overlap): to protect the train from overrunning the route, a certain distance after the end of the route should be detected vacant.

Any unsafe combinations of movements are usually excluded by several of the requirements listed above, so the specifications often have redundancies.

Interlocking systems have been designed and used since around 1850, and were at first implemented using mechanical locks, later using relay electronics (starting from the 1920s), and finally using computer software (starting from the 1980s), which is the most common method used today. The systems and specifications vary greatly between different countries in terms of principles, requirements, and implementation. For a more detailed account of railway operations, including a comparison of British, German, and North American concepts, see *Railway Operation and Control* by Jörn Pachl [131].

Figure 1.5 demonstrates how an interlocking works in principle, by showing a sequence of events that results in moving a train ahead by one route.

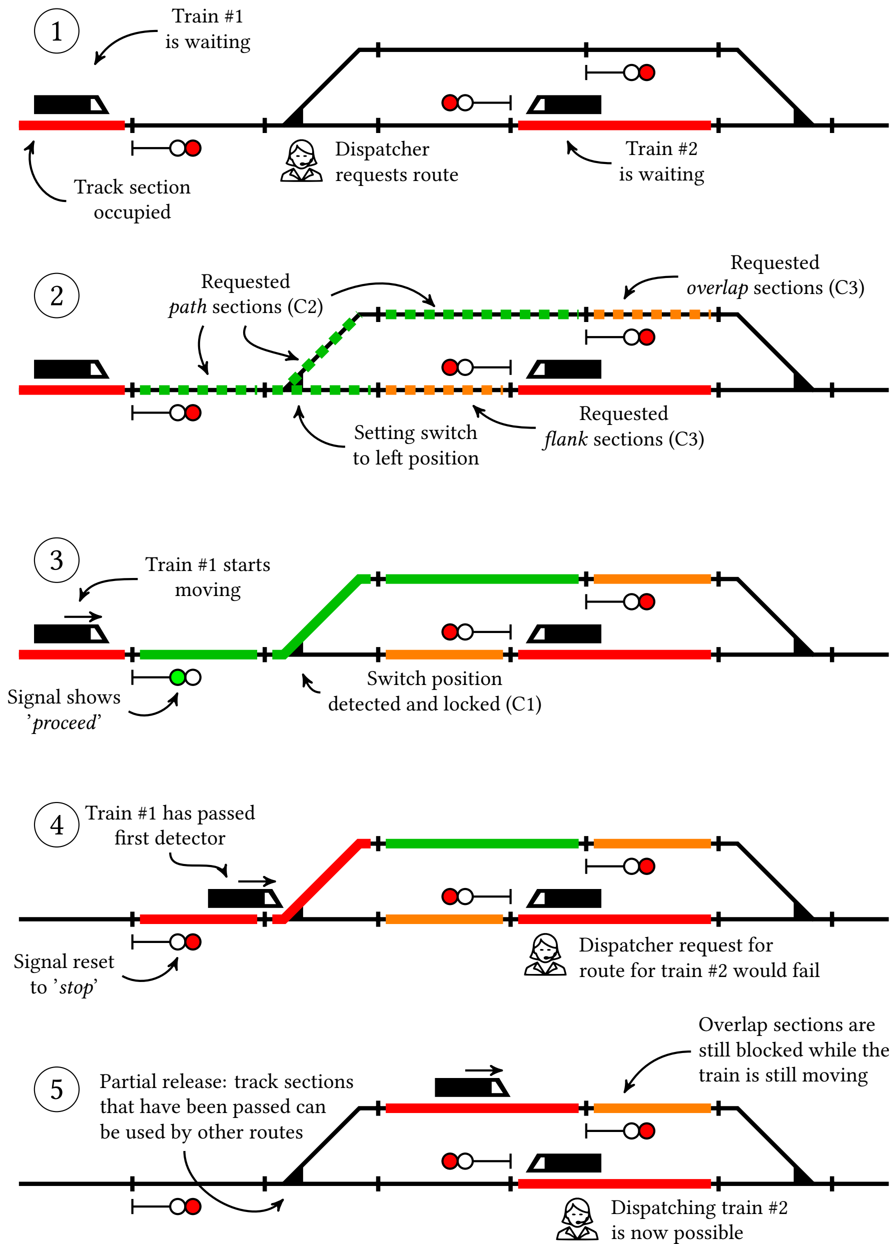


Figure 1.5: A sequence of events resulting in moving a train ahead one elementary route, exemplifying route setting, allocation of resources, and partial release.

1.3 Software tool support for railway infrastructure design

Computer software tools can have an important supporting role in the complex design processes for railway infrastructure, both for engineering analysis and verification work, and for cross-discipline coordination and project management. The use of software for railway infrastructure engineering tasks has in many companies not advanced beyond computerized drafting. The main tools in use in railway engineering today are:

- **Drafting** tools, often called CAD (computer-aided design or drafting), provide efficient means for producing geographical or schematic drawings of a construction project.

  Today, these are being extended to so-called BIM, Building Information Management, which typically means extending the CAD model to include 3D drawings of all disciplines for realistic visualization, and to include semantic data on components involved in the construction process.

  Examples of general CAD programs in use for all sub-disciplines include Autodesk AutoCAD and Bentley MicroStation. For some disciplines, there are also domain-specific CAD programs available, most notably for railway track design which is supported by Trimble NovaPoint, AKG Vestra, Card 1, and more. Signalling and interlocking design is supported by WSP ProSig and RailCOMPLETE.

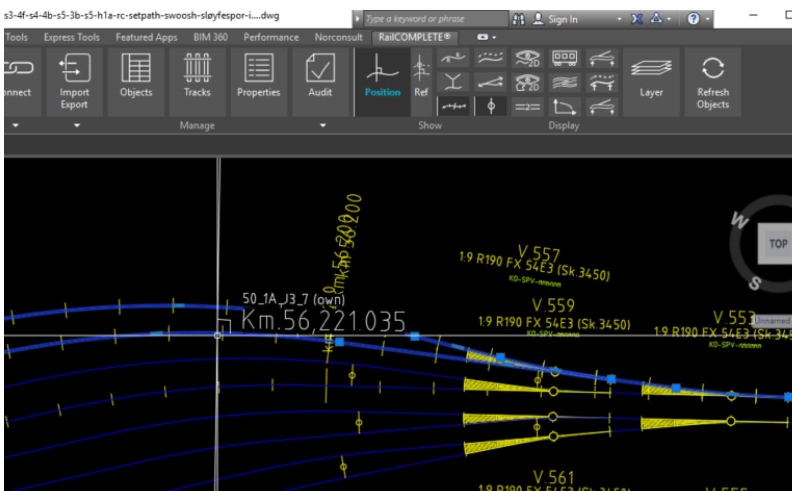


Figure 1.6: The RailCOMPLETE CAD tools extends Autodesk AutoCAD with railway-specific drafting and modelling capabilities.

- **Databases and models** of railway infrastructure and related information are used to store and transfer infrastructure data between companies, and between engineering, maintenance and operations sub-organizations. Typically, each national railway has a database model and a central database to store information about their railway network for engineering, maintenance, and operational planning. Examples include Ariane/Gaïa in the French SNCF railways, PlanProML in the German DB railways, and Banedata in the Norwegian Bane NOR railways.

  Some efforts on international standards for data models are gaining traction, such as railML, RailTopoModel, EuLynx, and IFC, all of which are aimed at improving integration between software tools, and data exchange between infrastructure managers and contractors across different countries.

  The railML format (see Figure 1.7) is an XML based language for data exchange of railway designs, developed by an international standardization committee, and was used in the work with this thesis to exchange infrastructure data between programs. railML consists of sub-schemas for time table, rolling stock, and infrastructure. The infrastructure schema is organized with a list of tracks at the top level of the hierarchy. Tracks contain sub-elements for (1) movable track elements, such as switches, crossings, and derailers, (2) trackside elements such as signals, detectors, and balises, (3) track geometry, such as radius and gradient, and (4) operational status, such as country borders, electrification, platform adjacency, and much more. Ends of tracks, along with switches and crossings, are considered *nodes* which can be connected to each other by mutually cross-referencing each other by name (using the XML attribute `id`). The recent railML version 3 now also contains a sub-schema for interlocking specifications.

```
- <infrastructure id="inf01">
  - <tracks>
    - <track id="tr01" name="track a02" type="secondaryTrack" mainDir="none">
      - <trackTopology>
        - <trackBegin pos="0" id="tr01_tb">
            <bufferStop id="tr01_bs01"/>
          </trackBegin>
      - <trackElements>
        - <platformEdges>
            <platformEdge id="tr01_pe01" name="Gleis 2" xml:lang="de" pos="200" dir="up"
            height="550" length="200"/>
          </platformEdges>
        </trackElements>
      - <ocsElements>
        - <signals>
          - <signal id="tr01_si01" code="68N1" pos="450" absPos="450" dir="up" function=
            ocpStationRef="ocp02">
              <etcs switchable="false" level_2="true"/>
            </signal>
          </signals>
```

Figure 1.7: The railML XML format is hierarchically structured, the infrastructure element contains tracks, and tracks contain trackside elements such as signals, and track geometry features such as radius and gradient.

- **Analysis** tools are typically specific to a sub-discipline. A major category of analysis tools is the capacity and time tabling tools. These tools use detailed data from the infrastructure, rolling stock, and time table domains to analyse changes to time table and stochastic effects of delays and congestion. Examples of capacity and time table analysis tools include VIA LUKS (see Figure 1.8), OpenTrack, and RMCon RailSys.

  Examples of other analysis domains include Sicat Master for catenary power line analysis, and Prover Trident for formal verification of interlocking implementations.

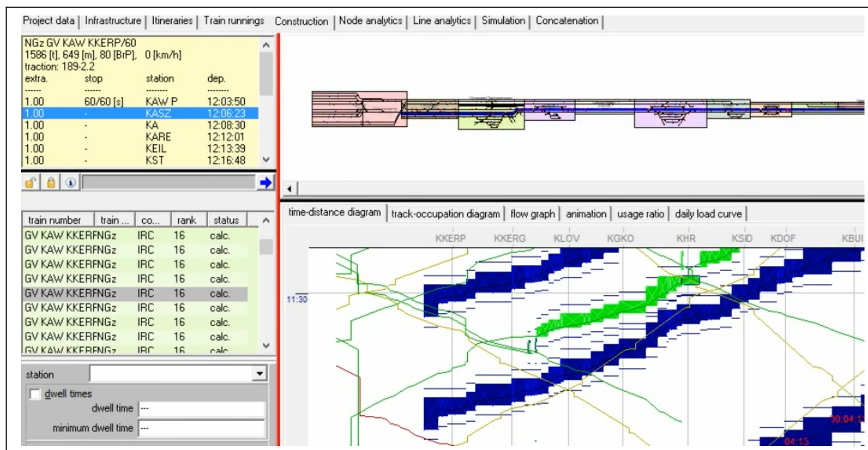


Figure 1.8: The LUKS capacity analysis tool offers analytical models and simulation models on comprehensive infrastructure and time table data.

1.4 Automated reasoning and formal methods

The sub-discipline of computer science called formal methods is concerned with formulating statements about computer programs and systems using mathematical language, and developing automated reasoning tools that can aid in verifying that systems adhere to the specifications written in this mathematical language. Most notably, formal methods are used in industrial software development for safety critical applications (for safety reasons), and in electronic hardware design (for economical reasons), see [36, 18].

Because both the design principles and the details of regulatory compliance and standards compliance in railway infrastructure design are complex and highly country specific, it might not be cost-effective to write custom software for each analysis task that might be useful for the end user. However, declarative logic-

based programming allows development of software where logical specifications are separated from the algorithm that performs the analysis. This means re-casting analysis problems from specific-purpose algorithms to the *verification* problem, as known from formal methods of computer science. This method consists of splitting an analysis task into *models* and *properties* which operate in a certain *logic*, which typically has a more general purpose than the specific problem being solved.

The fact that properties to be verified are considered an input to the *solver* program at the same level as the models themselves, means that the resulting software is more flexible in what it can be used for.

Another feature of the formal methods style of analysis is that the there are many logics which have been thoroughly studied for how to solve equations or answer queries about them efficiently. The formal logics that we have used in this thesis have advanced, efficient solvers:

- **Datalog**, a restricted form of logic programming which can only express tractable, terminating computations, has several highly optimized solvers, e.g. Soufflé [83], RDFox [122], and XSB Prolog [160].
- **SAT**, the Boolean satisfiability problem, consists of checking whether assigning truth values to the propositions of a propositional logic formula can make the formula true. This is in general a hard problem, but many solvers exists which solve real-world problems very efficiently, e.g. MiniSat [52] and Glucose [7].
- **SMT**, satisfiability modulo theories, integrates solvers for equations over reals, integers, and data structures such as arrays, with a SAT solver. Several solvers are popular for a wide range of problem solving, e.g. Z3 [43] and MathSAT [33].

Several such solver categories have annual competitions for fastest solvers on different kinds of benchmark problems, and significant effort has been put into solving arbitrary formulas and equations. For computationally hard problems, it is often much easier to model a problem in a suitable logic and use a well-developed solver than to create a special-purpose analysis program from first principles.

The tools presented in this thesis use automated reasoning and formal methods to de-couple highly specific domain logic from software development, so that engineers in different countries can use the same tools. By expressing properties in a suitable logic and using an efficient solver for that logic, it is possible to develop analysis tools that produce their results very fast, often with performance comparable to special-purpose algorithms. The efficient working of analysis tools can be highly important for using a tool in practice.

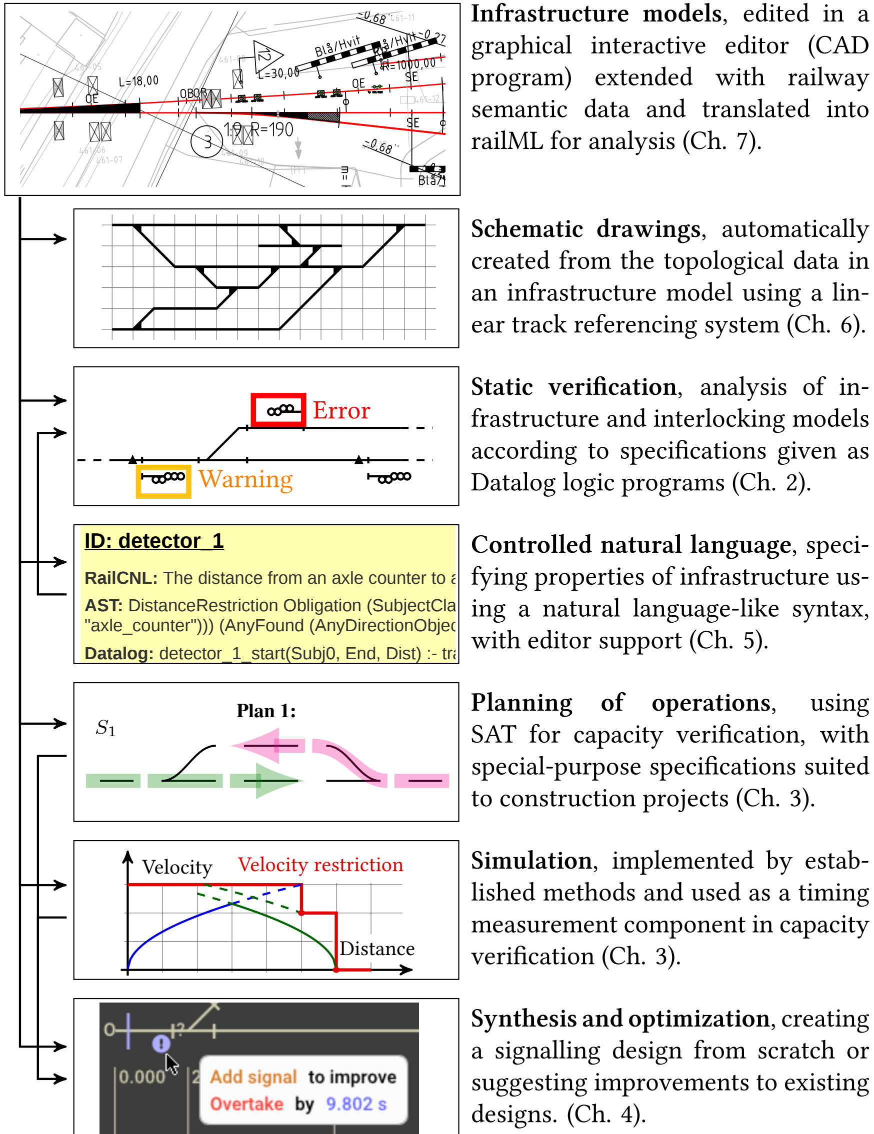
Railway control systems and signalling designs are a fertile ground for formal methods. See [16, 56] for an overview of various approaches and pointers to the literature, applying formal methods in various phases of railway design. For a slightly more dated state-of-the-art survey, see [77]. In particular, safety of interlockings has been intensively formalized and studied, using for instance VDM [61] and the

B-method, resp. Event-B [98]. Model checking has proved particularly attractive for tackling the safety of interlocking, and various model checkers and temporal logics have been used, cf. e.g. [26, 172, 54] [134, 112, 64, 54]. Critically evaluating practicality, [57] investigated applicability of model checking for interlocking tables using NuSMV and Spin, two prominent representatives of BDD-based symbolic model checking, and explicit state model checking, respectively. The research shows that interlocking systems of realistic size are currently out of reach for both flavors of general purpose model checkers. To mitigate the state-space explosion problem, *bounded* model checking has been extensively used for for interlockings. Instead of attempting an exhaustive coverage of the state-space, symbolically or explicitly, bounded model checking analysis (the behaviour of) a given system only up to a given bound (which is raised incrementally in case analysing a problem instance is inconclusive). This restriction allows to use SAT solving techniques in the analysis. The standard BMC references are [13, 35], however the techinque was already suggested for railway verification in [65]. [76] uses BMC on a variant of linear temporal logic (LTL) for safety property specification and employs so-called k-induction. The work of [173] investigates how to exploit domain-specific knowledge about interlocking verification to obtain good variable orderings when encoding the systems to be verified in a BDD-based symbolic model checker. An influential technology is the tool-based support for verified code generation for railway interlockings from Prover AB Sweden [19]. Prover is an automated theorem prover, using Stålmarck's method [158] of tautology checking.

Most of the approaches that can be found in the formal methods literature are concerned with the correct implementation of the control system's logic, and usually take the station layout as a given. In contrast, this thesis presents tools for working out these station layouts by placing signals, detectors, and related equipment onto the track plan.

1.5 Research contributions

The following overview shows the components of our railway design tool chain that we have developed and which is described in the following chapters.



Infrastructure models, edited in a graphical interactive editor (CAD program) extended with railway semantic data and translated into railML for analysis (Ch. 7).

Schematic drawings, automatically created from the topological data in an infrastructure model using a linear track referencing system (Ch. 6).

Static verification, analysis of infrastructure and interlocking models according to specifications given as Datalog logic programs (Ch. 2).

Controlled natural language, specifying properties of infrastructure using a natural language-like syntax, with editor support (Ch. 5).

Planning of operations, using SAT for capacity verification, with special-purpose specifications suited to construction projects (Ch. 3).

Simulation, implemented by established methods and used as a timing measurement component in capacity verification (Ch. 3).

Synthesis and optimization, creating a signalling design from scratch or suggesting improvements to existing designs. (Ch. 4).

The overview above is organized like the end-user software tools that we have developed, with closely dependent modules listed together. The rest of this text is however organized thematically:

- **Chapter 2**: static analysis, which is treated with Datalog logic programming, and integrated into a CAD program.

  This chapter contributes to goals A and B.

- **Chapter 3**: dynamic analysis for assessing capacity of a signalling design, which is treated as a SAT problem with simulation testing.

  This chapter contributes to goals A and B.

- **Chapter 4**: synthesis of signalling designs, treated using SAT and numerical optimization with the dynamic analysis techniques as a basis.

  This chapter contributes to goal C.

- **Chapter 5**: controlled natural language specification of railway regulations, treated using the Grammatical Framework programming language to define a domain-specific grammar and translating the resulting statements into the relevant formal logic.

  This chapter contributes to goal A and B.

- **Chapter 6**: drawing schematic railway plans by using SAT and optimization.

  This chapter contributes to goal B.

- **Chapter 7**: integrating a data model of railway infrastructure and various analysis tools into the interactive modelling tools used by railway engineers in practice (CAD tools).

  This chapter contributes to goal B.

Finally, some concluding remarks about our findings and developments and the way forward for taking railway engineering into the future can be found in Chapter 8.

2 Static analysis in infrastructure verification

Railway construction projects are heavy processes that integrate various fields, engineering disciplines, different companies, stakeholders, and regulatory bodies. When working out railway designs a large part of the work is repetitive, involving routine checking of consistency with regulations, writing tables, and coordinating disciplines. Many of these manual checks are simple enough to be automated. The repetition comes from the fact that even small changes in station layout and interlocking may require thorough (re-)investigation to prove that the designs remain internally consistent and still adhere to the rules and regulations of the national (and international) rail administration agencies.

With the purpose of increasing the degree of automation, this chapter presents results on integrating formal methods into the railway design process by the following means:

- Formalizing rules governing track and signalling layout, and interlocking.
- Using the standardized *railway markup language*, railML[1], as basis and exchange format for the formalization.
- Modelling the concepts describing a railway design in the logic of Datalog; and developing an automated generation of the model from the railML representation.
- Developing a prototype tool and integrating it in existing railway CAD software.

We illustrate the logical representation of signalling principles and show how they can be implemented and solved efficiently using the Datalog style of logic programming [165]. We also show the integration with existing railway engineering workflow by using CAD models directly. This enables us to verify compliance with regulations continuously as the design process changes the station layout and interlocking. Based on railML [121] as intermediary language, our results can be easily adopted by anyone who uses this international standard.

The approach presented in here could be applied also to other engineering disciplines, such as catenary power lines, track works, and others, which have similar design regulations and often make use of a similar CAD environment. However, we use the signalling and interlocking design process as an example and show how it can be improved by automation using formal methods.

[1] railML.org: `https://www.railml.org/`

The work uses as case study the software and the design (presently under development) used in the *Arna-Fløen* upgrade project,[2] a major infrastructure activity of the Norwegian railway system, with planned completion in 2020. The Arna train station is located on Northern Europe's busiest single-track connection (between Arna and Bergen), which is being extended to a double-track connection. Thus, the train station is currently undergoing an extensive overhaul, including significant new tunnel constructions and specifically a replacement of the entire signalling and control system. The case study is part of an ongoing project in Anacon AS (now merged with Norconsult), a Norwegian signalling design consultancy. It is used to illustrate the approach, test the implementation, and to verify that the tool's performance is acceptable for interactive work within the CAD software.

2.1 Verification artifacts

The signalling design process results in a set of documents which can be categorized into (a) track and signalling component layout, (b) interlocking specification, and (c) automatic train control specification. The first two categories are considered here.

2.1.1 Track and signalling component layout

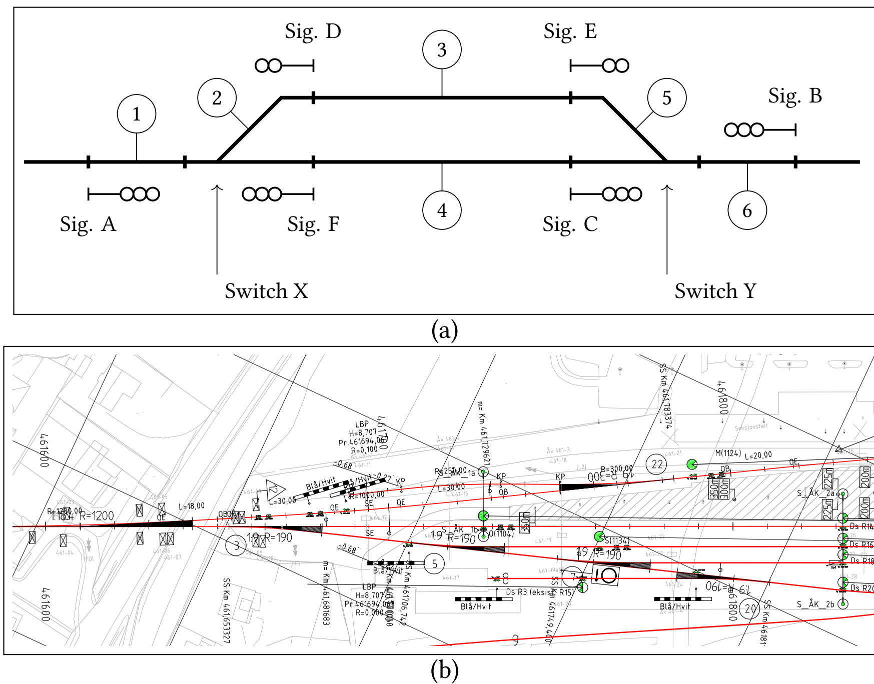
Railway construction projects rely heavily on *computer aided design* (CAD) tools to map out railway station layouts. The various disciplines within a project, such as civil works, track works, signalling, or catenary power lines, work with *coordinated CAD models*. These CAD models contain a major part of the work performed by engineers, and are a collaboration tool for communication between disciplines. The signalling component layout is worked out by the signalling engineers as part of the design process. Signals, train detectors, derailers, etc., are drawn using symbols in a 2D geographical CAD model. An example of a layout drawing made from a CAD model is given in Figure 2.1.

Track layout details, which are input for the signalling design, are often given by a separate division of the railway project. At an early stage and working at a low level of detail, the signalling engineer may challenge the track layout design, and an iterative process may be initiated.

2.1.2 Interlocking specification

An interlocking is an interconnection of signals and switches to ensure that train movements are performed in a safe sequence [131]. Interlocking is performed electronically so that, e.g., a green light (or, more precisely, the *proceed aspect*) communicating the movement authority required for a train to travel through a station

[2] http://www.jernbaneverket.no/Prosjekter/prosjekter/Arna---Bergen



(a)

(b)

Figure 2.1: (a) Example *schematic* construction drawing. (b) Cut-out from 2D geographical CAD model (construction drawing) of preliminary design of the Arna station signalling.

can only be lit by the interlocking controller under certain conditions. Conditions and state are built into the interlocking by relay-based circuitry or by computers running interlocking software. Most interlocking specifications use a *route-based tabular* approach, which means that a train station is divided into possible *routes*, which are paths that a train can take from one signal to another. These signals are called the *route entry signal* and *route exit signal*, respectively. An *elementary route* contains no other signals in-between. The main part of the interlocking specification is to tabulate all possible routes and set conditions for their use. Typical conditions are:

- *Switches* must be positioned to guide the train to a specified route exit signal.
- *Train detectors* must show that the route is free of any other trains.
- *Conflicting routes*, i.e. routes which make use of the same track segments, including segments inside safety margins, must not be in use.

Route	Start	End	Sw. pos	Detection sections	Conflicts
AC	A	C	X right	1, 2, 4	AE, BF
AE	A	E	X left	1, 2, 3	AC, BD
BF	B	F	Y left	4, 5, 6	AC, BD
BD	B	D	Y right	3, 5, 6	AE, BF

Figure 2.2: Example of a *tabular interlocking*, showing available routes and their conditions.

2.2 Logic programming and knowledge-base systems

In order to automatically verify rules and regulations on these railway infrastructure model artifacts, we need a computer program which can check each property for violations with the given model as input.

A straight-forward approach to making such a program could be to create some search function on the graph implicit in the track network. This procedure should allow, for example, to find the nearest object of a given type, or to find all paths between two points. Then we would describe a checking procedure for each rule. Consider for example the *home signal* regulation, introduced as Property 1 in Section 2.2.5 below, which says "A *home main signal* shall be placed at least 200 m in front of the first controlled, facing switch in the entry train path.". Checking such a property can be done by iterating over tracks, locating station boundaries, starting a search function to locate the relevant facing switches, starting another search backwards to check that there is a home signal, and so on. The amount of code required to do this in a mainstream programming language can become large, and this code is often very specific to a given railway administration.

Better suited to manage the large amounts of code required for a large number of rules, is *logic programming*, which allows rule descriptions that are much closer to the original specifications than in a mainstream programming language.

2.2.1 Logic programming

Logic programming [124] is a family of programming languages based on formal logic. Logic programs are *declarative*, i.e. they describe properties of the solution of a problem rather than a calculation procedure for finding the solution. This separates the concerns of expressing rules about railway systems from the algorithms required to do automatic analysis. This separation allows one to systematically maintain a large set of rules, and decouple the tool implementation from the set of concepts, rules and expert knowledge that is specific to a railway administration.

We have successfully used the Datalog language [165], a subset of the more well-known Prolog language, for verifying many properties given as technical rules

and expert knowledge. It allows concise formulations of railway concepts, and queries can be efficiently calculated.

Ideally, we would like the railway engineers themselves, without much programming education, to be able to create and maintain the set of rules which is used for the verification. This separation of logic and algorithm is a step in this direction, because non-IT experts can work on the rules without considering how the calculations are implemented. However, the strict formalism and subtle semantics of logic programming are still a challenge for an inexperienced programmer.

Still we think that it is feasible for inexperienced logic programmers to do some of the maintenance of a rule base for the following reasons:

1. The most basic concepts, such as connectedness, distances, directions, etc., rarely need to be redefined, and may be specified by an expert programmer, and then reused.

2. Naming or documenting the basic concepts in a way that is understandable by railway engineers allows them to use these concepts without considering the actual definitions.

3. Rule formulations are often so succinct that they can be understood even without knowledge of the logic programming syntax.

4. Modification (updating) of rules, for example following a change in regulation from the railway administration, often preserves the structure of the specification, adding only a similar clause or the change of a numeric constant.

5. Templates for common rule structures can be given, so that implementing some types of rules becomes a matter of specifying e.g. only object types, directions, and distances.

We envision that a common rule base would be exchanged between all engineers working with a railway administration, and that the rule base would be worked out partly by software experts, partly by railway experts. Also, the rule base should be fairly constant, like the regulations, requiring an update frequency of perhaps once per year.

We do, however, concede that Datalog programming in general is outside what would be expected competency for a railway engineer. A higher-level domain-specific language including relevant constructs for a railway design knowledge base could improve the likelihood of railway engineers being successful in creating and maintaining the regulations. Even more, this language could allow each company and each engineer to experiment with encoding different design heuristics and expert knowledge to see the effects on the verification and the design. This idea is elaborated on in Chapter 5.

2.2.2 Datalog

Declarative logic programming is a programming language paradigm which allows clean separation of logic (meaning) and computation (algorithm). This section gives a short overview of Datalog concepts. See [165, 2, 124] for more details. In its most basic form a Datalog program is a database query, as in the SQL language, over a finite set of atoms which can be combined using conjunctive queries, i.e. expressions in the fragment of first-order logic which includes only conjunctions and existential quantification.

Conjunctive queries alone, however, cannot express the properties needed to verify railway signalling. For example, given the layout of the station with tracks represented as edges between signalling equipment nodes, graph reachability queries are required to verify some of the rules. This corresponds to computing the transitive closure of the graph adjacency relation, which is not expressible in first-order logic [99, Chap. 3].

Adding fixed-point operators to conjunctive queries is a common way to mitigate the above problem while preserving decidability and polynomial time complexity.

The Datalog language is a first-order logic extended with *least fixed points*. We define the Datalog language as follows: *Terms* are either constants (atoms) or variables. *Literals* consist of a *predicate* p with a certain arity n, along with terms corresponding to the predicate arguments, forming an expression like $p(\overrightarrow{a})$, where $\overrightarrow{a} = (a_1, a_2, \ldots, a_n)$. *Clauses* consist of a *head* literal and one or more *body* literals, such that all variables in the head also appear in the body. Clauses are written as

$$r_0(\overrightarrow{x}) \;\text{:-}\; \exists \overrightarrow{y} : r_1(\overrightarrow{x_1}, \overrightarrow{y_1}), r_2(\overrightarrow{x_2}, \overrightarrow{y_2}), \ldots, r_k(\overrightarrow{x_k}, \overrightarrow{y_k}),$$

with $\bigcup_{1 \le i \le k} \overrightarrow{x_i} = \overrightarrow{x}$ and $\bigcup_{1 \le i \le k} \overrightarrow{y_i} = \overrightarrow{y}$. Datalog uses the Prolog convention of interpreting identifiers starting with a capital letter as variables, and other identifiers as constants, e.g., the clause

$$\mathrm{a}(X, Y) \;\text{:-}\; \mathrm{b}(X, Z),\ \mathrm{c}(Z, Y)$$

has the meaning of

$$\forall x, y : ((\exists z : (b(x, z) \wedge c(z, y))) \rightarrow a(x, y))\,.$$

Clauses without body, which cannot then contain any variables, are called *facts*, those with one or more literals in the body are called *rules*. No nesting of literals is allowed. However, recursive definitions of predicates are possible. For example, let $edge(a, b)$ be a graph edge relation between vertices a and b. Graph searches can now be encoded by making a transitive closure over the edge relation:

$$\begin{aligned}
&\mathrm{path}(a, b) \;\text{:-}\; \mathrm{edge}(a, b).\\
&\mathrm{path}(a, b) \;\text{:-}\; \mathrm{edge}(a, x),\ \mathrm{path}(x, b).
\end{aligned}$$

In the railway domain, this can be used to define the *connected* predicate, which defines whether two objects are connected by railway tracks:

$$\begin{aligned}
\text{directlyConnected}(a,b) &\text{ :- } \text{track}(t),\ \text{belongsTo}(a,t),\ \text{belongsTo}(b,t). \\
\text{connected}(a,b) &\text{ :- } \text{directlyConnected}(a,b). \\
\text{connected}(a,b) &\text{ :- } \text{directlyConnected}(a,x),\ \text{connection}(x,c), \\
&\qquad \text{connected}(c,b).
\end{aligned}$$

Here, the *connection* predicate contains switches and other connection types. Further details of relevant predicates are given in the sections below.

Another common feature of Datalog implementations is to allow negation, with *negation as failure* semantics. This means that negation of predicates in rules is allowed with the interpretation that when the satisfiability procedure cannot find a model, the statement is false. To ensure termination and unique solutions, the negation of predicates must have a *stratification*, i.e. the dependency graph of negated predicates must have a topological ordering (see [165, Chap. 3] for details).

Datalog is sufficiently expressive to describe static rules of signalling layout topology and interlocking. For geometrical properties, it is necessary to take sums and differences of lengths, which requires extending Datalog with arithmetic operations. A more expressive language is required to cover all aspects of railway design, e.g. capacity analysis and software verification, but for the properties in the scope of this chapter, a concise, restricted language which ensures termination and short running times has the advantage of allowing tight integration with the existing engineering workflow.

2.2.3 Knowledge-base system

With Datalog as specification language, we build a *knowledge-base system* to perform the verification. A knowledge-base system consists of a set of facts and rules, along with an inference engine which answers queries by applying logical inference rules. For an introduction to knowledge-base systems in general, see [165, Chap. 3] or [149, Chap. 8 and 12]. We give here an overview of how we encode railway signalling properties as Datalog predicates, which in turn may be automatically checked for consistency. In our verification tool, we organize our knowledge base in the following manner:

1. **Input documents**: Predicate representation of input document, i.e. track layout and interlocking, are represented as *facts* which are converted from the railML representation stored and maintained in the CAD database by a CAD plug-in program.

2. **Derived concepts**: Predicate representation of derived concept *rules*, such as object properties, topological properties, and calculation of distances. A library

of general railway concepts and administration-specific concepts and definitions are kept in a rule base which is re-used between projects.

3. **Technical rules and expert knowledge**: Predicate representation of technical rules or expert knowledge as logic programming *rules*, which encode the administration-specific rules and expert knowledge that is checked and errors reported to the user by the verification tool.

4. **Inference engine**: A Datalog evaluation engine is used as inference engine; in our case the XSB Prolog tabled logic programming system [160].

Each of these aspects are described in more detail below.

2.2.3.1 Input documents

Each of the XML elements and attributes is translated into a corresponding predicate. An example of translating a railML *switch* element into predicate representation is given below.

```
<switch id='sw1'>                                      switch(sw1).
  <connection id='conn1' course='                      connection(conn1).
       left' orientation='outgoing'   →                belongsTo(sw1,conn1).
       />                                              course(conn1,left).
</switch>                                              orientation(conn1,outgoing).
```

2.2.3.2 Track and signalling objects layout in the railML format

Given a complete railML infrastructure document, we consider the set of XML elements in it that correspond to identifiable objects (this is the set of elements which inherit properties from the type `tElementWithIDAndName`). The set of all IDs which are assigned to XML elements form the finite domain of constants on which we base our predicates (IDs are assumed unique in railML).

$$\text{Atoms} := \{a \mid \text{element.}\mathtt{ID} = a\}\,.$$

We denote a railML element with $ID = a$ as element_a. All other data associated with an element is expressed as predicates with its identifying atom as one of the arguments, most notably the following:

- Element type (also called class in railML):

$$\begin{aligned}
\text{track}(a) &\leftarrow \text{element}_a \text{ is of type } \mathtt{track},\\
\text{signal}(a) &\leftarrow \text{element}_a \text{ is of type } \mathtt{signal},\\
\text{balise}(a) &\leftarrow \text{element}_a \text{ is of type } \mathtt{balise},\\
\text{switch}(a) &\leftarrow \text{element}_a \text{ is of type } \mathtt{switch}.
\end{aligned}$$

- Element name:
$$\text{name}(a,n) \leftarrow (\text{element}_a.\texttt{name} = n).$$

- Position and absolute position (elements inheriting from `tPlacedElement`):
$$\text{pos}(a,p) \leftarrow (\text{element}_a.\texttt{pos} = p), \quad a \in \text{Atoms}, p \in \mathbb{R},$$
$$\text{absPos}(a,p) \leftarrow (\text{element}_a.\texttt{absPos} = p), \quad a \in \text{Atoms}, p \in \mathbb{R}.$$

- Geographical coordinates (for elements inheriting from `tPlacedElement`):
$$\text{geoCoords}(a,q) \leftarrow (\text{element}_a.\texttt{geoCoords} = q), \quad a \in \text{Atoms}, q \in \mathbb{R}^3.$$

- Direction (for elements inheriting from `tOrientedElement`):
$$\text{dir}(a,d) \leftarrow (\text{element}_a.\texttt{dir} = d), \quad a \in \text{Atoms}, d \in \text{Direction},$$
where Direction $= \{up, down, both, unknown\}$, indicating whether the object is visible or functional in only one of the two possible travel directions, or both.

- Signal properties (for elements of type `tSignal`):
$$\begin{aligned}\text{signalType}(a,t) &\leftarrow (\text{element}_a.\texttt{type}{=}t),\\ &\quad t \in \{\text{main, distant, shunting, combined}\},\\ \text{signalFunction}(a,f) &\leftarrow (\text{element}_a.\texttt{function} = f),\\ &\quad a \in \text{Atoms}, f \in \{\text{home, intermediate, exit, blocking}\}.\end{aligned}$$
Consistency axioms would impose that *signalType* and *signalFunction* be applied only to *signal* elements:
$$\text{signalType}(a,t) \Rightarrow signal(a),$$
$$\text{signalFunction}(a,f) \Rightarrow signal(a).$$

These are only a few examples of predicates that are extracted from the railML document. The translator from railML to predicate form needs only to consider XML elements, attributes and sub-elements, not the specifics of railML and its type hierarchy. The complete structure of railML as such is carried over directly to the logic programming environment. The *switch* element is the object which connects tracks with each other and creates the branching of paths, see Figure 2.3. A switch belongs to a single track, but contains *connection* sub-elements which point to other connection elements, which are in turn contained in switches, crossings or track ends. For connections, we have the following predicates:

- Connection element and reference:
$$\text{connection}(a) \leftarrow \text{element}_a \text{ is of type } \texttt{connection},$$
$$\text{connection}(a,b) \leftarrow (\text{element}_a.\texttt{ref} = b).$$

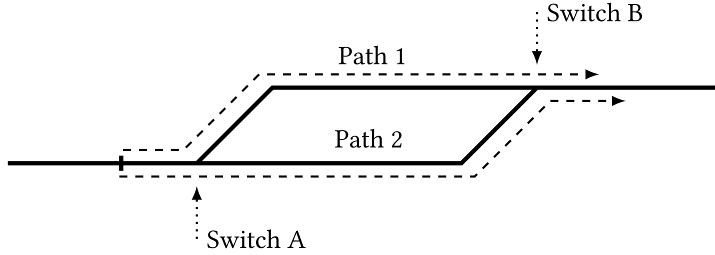


Figure 2.3: Switches give rise to branching paths

- Connection course and orientation:

$$\text{connectionCourse}(a, c) \leftarrow (\text{element}_a.\texttt{course} = c), c \in \{\text{left, straight, right}\}$$
$$\text{connectionOrientation}(a, o) \leftarrow (\text{element}_a.\texttt{orientation} = o),$$
$$a \in \text{Atoms}, o \in \{\text{outgoing, incoming}\}\,.$$

To encode the hierarchical structure of the railML document, a separate predicate encoding the parent/child relationship is added. This is required because the predicate representation does not implicitly contain the hierarchy of the XML representation, where elements are declared inside other elements.

- Object belongs to (e.g. a is a signal belonging to track b):

$$\text{belongsTo}(a, b) \leftarrow b \text{ is the closest XML ancestor of } a \text{ whose element type inherits from } \texttt{tElementWithIDAndName}.$$

2.2.3.3 Interlocking

An XML schema for tabular interlocking specifications is now included in version 3 of railML, but the interlocking encoding below was developed before railML 3 and is instead based on the preliminary railML work described in [21]. We give some examples of how this schema is translated into predicate form:

- Train route with given direction d, start point a, and end point b ($a, b \in$ Atoms, $d \in$ Direction):

$$\text{trainRoute}(t) \leftarrow \text{element}_t \text{ is of type } \texttt{route}$$
$$\text{start}(t, a) \leftarrow (\text{element}_t.\texttt{start} = a)$$
$$\text{end}(t, b) \leftarrow (\text{element}_t.\texttt{end} = b)$$

- Conditions on detection section free (a) and switch position (s, p):

$$\text{detectionSectionCondition}(t, a) \leftarrow (a \in \text{element}_t.\texttt{sectionConditions}),$$
$$\text{switchPositionCondition}(t, s, p) \leftarrow ((s, p) \in \text{element}_t.\texttt{switchConditions}).$$

2.2.4 Derived concepts representation

Derived concepts are properties of the railway model which can be defined independently of the specific station. A library of these predicates is needed to allow concise expression of the rules to be checked.

2.2.4.1 Object properties

Properties related to specific object types which are not explicitly represented in the layout description, such as whether a switch is *facing* in a given direction, i.e. if the path will branch when you pass it:

- Switch facing or trailing ($a \in$ Atoms, $d \in$ Direction):

$$\text{switchFacing}(a,d) \leftarrow \exists c,o : \text{switch}(a) \wedge \text{switchConnection}(a,c) \wedge \\ switchOrientation(c,o) \wedge orientationDirection(o,d).$$

$$\text{switchTrailing}(a,d) \leftarrow \neg\text{switchFacing}(a,d)$$

2.2.4.2 Topological and geometric layout properties

Predicates describing the topological configuration of signalling objects and the train travel distance between them are described by predicates for **track connection** (predicate connected(a,b)), **directed connection** (predicate following(a,b,d)), **distance** (predicate distance(a,b,d,l)), etc. The track connection predicate is defined as:

- There is a **track connection** between object a and b ($a,b \in$ Atoms):

$$\text{directlyConnected}(a,b) \leftarrow \exists t : \text{track}(t) \wedge \text{belongsTo}(a,t) \wedge \text{belongsTo}(b,t),$$

$$\text{connected}(a,b) \leftarrow \text{directlyConnected}(a,b) \vee (\exists c_1,c_2 : \text{connection}(c_1,c_2) \wedge \\ \text{directlyConnected}(a,c_1) \wedge \text{connected}(c_2,b)).$$

- There is a **directed connection** between object a and b ($a,b \in$ Atoms, $d \in$ Direction, $p_a, p_b \in \mathbb{R}$):

$$\text{directlyFollowing}(a,b,d) \leftarrow \text{directlyConnected}(a,b) \wedge \\ \text{position}(a,p_a) \wedge \text{position}(b,p_b) \wedge \\ ((d = \text{up} \wedge p_a < p_b) \vee (d = \text{down} \wedge p_a > p_b))$$

$$\text{following}(a,b,d) \leftarrow \text{directlyFollowing}(a,b,d) \vee \\ \exists c_1,c_2 : \text{connection}(c_1,c_2) \wedge \text{directlyFollowing}(a,c_1,d) \\ \wedge \text{following}(c_2,b,d)$$

- The **distance** (along track) in a given direction between object a and b ($a, b \in$ Atoms, $d \in$ Direction, $p_a, p_b, l \in \mathbb{R}$):

$$\begin{aligned}\text{directDistance}(a,b,d,l) \leftarrow\ & \text{directlyFollowing}(a,b,d) \wedge \\ & \text{position}(a,p_a) \wedge \text{position}(b,p_b) \\ & \wedge\, l = |p_b - p_a|\end{aligned}$$

$$\begin{aligned}\text{distance}(a,b,d,l) \leftarrow\ & \text{directDistance}(a,b,d,l) \vee \\ & \exists c_1, c_2, l_1, l_2 : \text{connection}(c_1,c_2) \\ & \wedge\, \text{directDistance}(a,c_1,d,l_1) \\ & \wedge\, \text{distance}(c_2,b,d,l_2) \wedge l = l_1 + l_2\end{aligned}$$

- Object is located **between** a and b ($a, x, b \in$ Atoms, $d \in$ Direction):

$$\text{between}(a,x,b,d) \leftarrow \text{following}(a,x,d) \wedge \text{following}(x,b,d)$$

$$\text{between}(a,x,b) \leftarrow \exists d : \text{between}(a,x,b,d)$$

- A path between a and b **overlaps** with a path between c and d ($a, b, c, d \in$ Atoms):

$$\text{overlap}(a,b,c,d) \leftarrow \exists e : \text{between}(a,e,b) \wedge \text{between}(c,e,d)$$

2.2.4.3 Interlocking properties

Predicates such as existsPathWithoutSignal(a, b) which defines the method for finding elementary routes, and existsPathWithDetector(a, b) for finding adjacent train detectors, will be used as building blocks for the interlocking rules. We show here a recursive rule used for finding elementary routes:

- Signals a and b have a path between them without any other signals in between:

$$\begin{aligned}\text{existsPathWithoutSignal}(a,b,d) \leftarrow\ & \text{following}(a,b,d) \wedge \\ & (\neg(\exists x : \text{signal}(x) \wedge \text{between}(a,x,b)) \vee \\ & (\exists x : \text{between}(a,x,b) \wedge \text{existsPathWithoutSignal}(a,x,d) \wedge \\ & \text{existsPathWithoutSignal}(x,b,d)).\end{aligned}$$

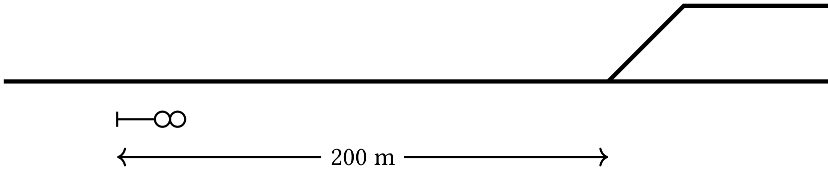
2.2.5 Rule violations representation

With the input documents represented as facts, and a library of derived concepts, it remains to define the technical rules to be checked. All technical rules presented herein are based on the Norwegian infrastructure manager's regulations[3]. The goal of the consistency checking is to confirm that no inconsistencies exist, in which

[3] Bane NOR: Teknisk regelverk, `http://trv.banenor.no/`

case no further information is required, or to find inconsistencies and present them in a way that allows the user to understand the error and to adjust their design accordingly. Rules are therefore expressed negatively, as rule *violations*, so that a query corresponding to the rule is empty whenever the rule is consistent with the design, or the query contains counterexamples to the rule when they exist. Some examples of technical rules representing conditions of the railway station layout are given below.

Property 1 (Layout: Home signal). *A* home main signal *shall be placed at least 200 m in front of the first controlled, facing switch in the entry train path.*



Property 1 may be represented in the following way:

$$\begin{aligned}\text{isFirstFacingSwitch}(b, s) \leftarrow\ & \text{stationBoundary}(b) \wedge \text{facingSwitch}(s) \wedge \\ & \neg(\exists x : \text{facingSwitch}(x) \wedge \text{between}(b, x, s)),\end{aligned}$$

$$\begin{aligned}\text{ruleViolation}_1(b, s) \leftarrow\ & \text{isFirstFacingSwitch}(b, s) \wedge \\ & (\neg(\exists x : \text{signalFunction}(x, \text{home}) \wedge \text{between}(b, x, s)) \vee \\ & (\exists x, d, l : \text{signalFunction}(x, \text{home}) \wedge \\ & \wedge\, \text{distance}(x, s, d, l) \wedge l < 200).\end{aligned}$$

Checking for rule violations can be expressed as:

$$\exists b, s : \text{ruleViolation}_1(b, s),$$

which in Datalog query format becomes `ruleViolation1(B,S)?`.

Property 2 (Layout: Minimum detection section length). *No train detection section shall be shorter than 21 m. I.e., no train detectors should be separated with less than 21 m driving distance.*

This property is represented as follows:

$$\begin{aligned}\text{ruleViolation}_2(a, b) \leftarrow & \exists d, l : \text{trainDetector}(a) \wedge \text{trainDetector}(b) \wedge \\ & \text{distance}(a, b, d, l) \wedge l < 21.0.\end{aligned}$$

Property 3 (Layout: Exit main signal). *An exit main signal shall be used to signal movement exiting a station.*

This property can be elaborated into the following rules:

- No path should have more than one exit signal:

$$\begin{aligned}\text{ruleViolation}_3(s) \leftarrow & \exists d : \text{signalType}(s, \text{exit}) \wedge \text{following}(s, s_o, d) \wedge \\ & \neg \text{signalType}(s_0, \text{exit}).\end{aligned}$$

- Station boundaries should be preceded by an exit signal:

$$\begin{aligned}\text{exitSignalBefore}(x, d) \leftarrow & \exists s : \text{signalType}(s, \text{exit}) \wedge \text{following}(s, x, d) \\ \text{ruleViolation}_3(b) \leftarrow & \exists d : \text{stationBoundary}(b) \wedge \neg \text{exitSignalBefore}(b, d).\end{aligned}$$

A basic property of tabular interlockings is that each consecutive pair of main signals normally has an elementary train route associated with it, i.e.:

Property 4 (Interlocking: Elementary routes). *A pair of consecutive main signals should be present as a route in the interlocking.*

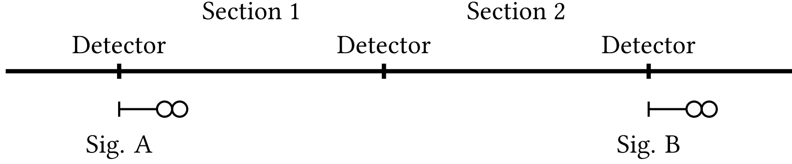
This can be represented as follows:

$$\begin{aligned}\text{defaultRoute}(a, b, d) \leftarrow\ & \text{signalType}(a, \text{main}) \wedge \text{signalType}(b, \text{main}) \wedge \\ & \text{direction}(a, d) \wedge \text{direction}(b, d) \wedge \\ & \text{following}(a, b, d) \wedge \text{existsPathWithoutSignal}(a, b, d),\end{aligned}$$

$$\begin{aligned}\text{ruleViolation}_4(a, b, d) \leftarrow\ & \text{defaultRoute}(a, b, d) \wedge \\ & \neg(\exists r : \text{trainRoute}(r) \wedge \text{trainRouteStart}(r, a) \wedge \text{trainRouteEnd}(r, b)).\end{aligned}$$

This type of rule is not absolutely required for a railway signalling design to be valid and safe. Some rules are hard constraints, where violations may be considered to be errors in the design, while other rules are soft constraints, where violations may suggest that further investigation is recommended. This is relevant for the counterexample presentation section below.

Property 5 (Interlocking: Track clear on route). *Each pair of adjacent train detectors defines a track detection section. For any track detection sections overlapping the route path, there shall exist a corresponding condition on the activation of the route.*



Tabular interlocking:

Route	Start	End	Sections must be clear
AB	A	B	1, 2

Property 5 can be represented as follows:

$$\begin{aligned}\text{existsPathWithDetector}(a, b) \leftarrow & \exists d : \text{following}(a, b, d) \wedge \text{trainDetector}(x) \wedge \\ & \text{between}(a, x, b).\end{aligned}$$

$$\begin{aligned}\text{adjacentDetectors}(a, b) \leftarrow & \text{trainDetector}(a) \wedge \text{trainDetector}(b) \wedge \\ & \neg\text{existsPathWithDetector}(a, b),\end{aligned}$$

$$\begin{aligned}\text{detectionSectionOverlapsRoute}(r, d_a, d_b) \leftarrow & \ \text{trainRoute}(r) \wedge \\ & \text{start}(r, s_a) \wedge \text{end}(r, s_b) \wedge \\ & \text{adjacentDetectors}(d_a, d_b) \wedge \text{overlap}(s_a, s_b, d_a, d_b),\end{aligned}$$

$$\begin{aligned}\text{detectionSectionCondition}(r, d_a, d_b) \leftarrow & \ \text{detectionSectionCondition}(c) \wedge \\ & \text{belongsTo}(c, r) \wedge \text{belongsTo}(d_a, c) \wedge \text{belongsTo}(d_b, c).\end{aligned}$$

$$\begin{aligned}\text{ruleViolation}_5(r, d_a, d_b) \leftarrow & \\ & \text{detectionSectionOverlapsRoute}(r, d_a, d_b) \wedge \\ & \neg\text{detectionSectionCondition}(r, d_a, d_b).\end{aligned}$$

Property 6 (Interlocking: Flank protection). *A train route shall have flank protection.*

For each switch in the route path and its associated position, the paths starting in the opposite switch position defines the *flank*. Each flank path is terminated by the first flank protection object encountered along the path. The following objects can give flank protection:

1. *Main signals*, by showing the *stop* aspect.

2. *Shunting signals*, by showing the *stop* aspect.

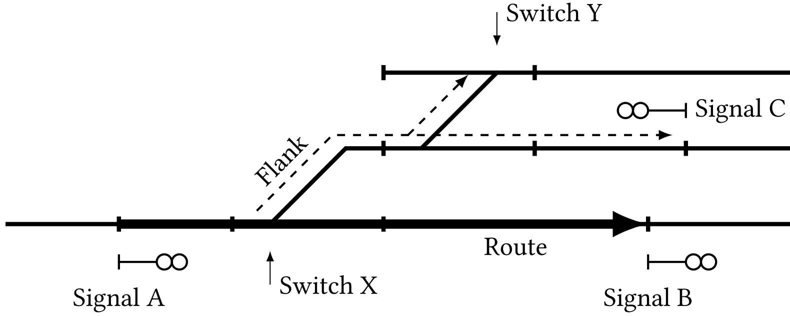


Figure 2.4: The dashed path starting in switch X must be terminated in all branches by a valid flank protection object, in this case switch Y and signal C. (Property 6)

3. *Switches*, by being controlled and locked in the position which does not lead into the path to be protected.

4. *Derailers*, by being controlled and locked in the derailing state.

An example situation is shown in Figure 2.4. While the indicated route is active (A to B), switch X needs flank protection for its left track. Flank protection is given by setting switch Y in right position and setting signal C to *stop*. Property 6 can be elaborated into the following rules:

- All flank protection objects should be eligible flank protection objects, i.e. they should be in the list of possible flank protection objects, and have the correct orientation (the *flankElement* predicate contains the interlocking facts):

$$\begin{aligned}\text{flankProtectionObject}(a, b, d) \leftarrow &((\text{signalType}(a, \text{main}) \wedge \text{dir}(a, d)) \vee \\ &(\text{signalType}(a, \text{shunting}) \wedge \text{dir}(a, d)) \vee \\ &\text{switchFacing}(a, d) \vee \\ &\text{derailer}(a)) \wedge \text{following}(a, b, d).\end{aligned}$$

$$\begin{aligned}\text{flankProtectionRequired}(r, x, d) \leftarrow\ &\text{trainRoute}(r) \wedge \text{start}(r, s_a) \wedge \\ &\text{end}(r, s_b) \wedge \text{switchOrientation}(x, o) \wedge \text{between}(s_a, x, s_b) \wedge \\ &\text{orientationDirection}(o, o_d) \wedge \text{oppositeDirection}(o_d, d).\end{aligned}$$

$$\begin{aligned}\text{flankProtection}(r, e) \leftarrow &\text{flankProtectionRequired}(r, x, d) \wedge \\ &\text{flankProtectionObject}(e, x, d).\end{aligned}$$

$$\begin{aligned}\text{ruleViolation}_6(r, e) \leftarrow &\text{flankElement}(r, e) \wedge \\ &\neg\text{flankProtection}(r, e).\end{aligned}$$

- There should be no path from a model/station boundary to the given switch, in the given direction, that does not pass a flank protection object for the route:

$$\begin{aligned}&\text{existsPathWithFlankProtection}(r,b,x,d) \leftarrow \\ &\qquad \text{flankElement}(r,e) \land \text{flankProtectionElement}(e,x,d) \land \\ &\qquad \text{between}(b,e,x).\end{aligned}$$

$$\begin{aligned}&\text{existsPathWithoutFlankProtection}(r,b,x,d) \leftarrow \\ &\qquad \neg\text{existsPathWithFlankProtection}(r,b,x,d) \lor \\ &\qquad (\text{between}(b,y,x) \land \neg\text{flankProtectionElement}(e,y,d) \land \\ &\qquad \text{existsPathWithoutFlankProtection}(r,b,y,d) \land \\ &\qquad \text{existsPathWithoutFlankProtection}(r,y,x,d)).\end{aligned}$$

$$\begin{aligned}&\text{ruleViolation}_6(r,b,x) \leftarrow \text{stationBoundary}(b) \land \\ &\qquad \text{flankProtectionRequired}(r,x,d) \land \text{following}(b,x,d) \land \\ &\qquad \text{existsPathWithoutFlankProtection}(r,b,x,d).\end{aligned}$$

2.3 Tool implementation

The XSB Prolog interpreter [160] was used as a back-end for the implementation as it offers tabled predicates which have the same characteristics as Datalog programs, while still allowing general Prolog expressions such as arithmetic operations.

The translation from railML to Datalog facts assumes that the document is valid railML, which may be checked with general XML schema validators, or a specialized railML validator.

2.3.1 Counterexample Presentation

When rule violations are found, the railway engineer will benefit from information about the following:

- Which rule was violated (textual message containing a reference to the source of the rule or a justification in the case of expert knowledge rules).
- Where the rule was violated (identity of objects involved).

Also, classification of rules based on e.g. *discipline* and *severity* may be useful in many cases. In the rule databases, this may be accomplished through the use of *structured comments*, similar to the common practice of including structured documentation in computer programs, such as JavaDoc (see Figure 2.5 for an example of how we do this). A program parses the structured comments and forwards corresponding queries to the logic programming solver. Any violations returned are associated with the information in the comments, so that the combination can be used to present a helpful message to the user. We implemented a prototype CAD add-on program for Autodesk AutoCAD (see Section 7.2).

```
%| rule: Home signal too close to first facing switch.
%| type: technical
%| severity: error
homeSignalBeforeFacingSwitchError(S,SW) :-
    firstFacingSwitch(B,SW,DIR),
    homeSignalBetween(S,B,SW),
    distance(S,SW,DIR,L), L < 200.
```

Figure 2.5: Structured comments on rule violation expression

2.3.2 Case study results

The rules concerning signalling layout and interlocking from *Bane NOR*[4] described above have been checked against the model (i.e., railML representation) of the Arna-Fløen project, which is an ongoing design project in Anacon AS (now merged with Norconsult AS). Each object was associated with one or more construction phases, which we call phase A and phase B, which also corresponds to two operational phases. The model that was used for the work with the Arna station (phase A and B combined) included 25 switches, 55 connections, 74 train detectors, and 74 signals. The interlocking consisted of 23 and 42 elementary routes in operational phase A and B respectively.

The Arna station design project and the corresponding CAD model has been in progress since 2013, and the method of integrating railML fragments into the CAD database, as described in Section 7.2, has been in use for more than one year. Engineers working on this model are now routinely adding the required railML properties to the signalling components as part of their CAD modelling process. This allowed a fully automated transfer of the railML station description to the verification tool. Several simplified models were made also for testing the correct functioning of the concept predicates and rule violation predicates. The rule collection consisted of 37 derived concepts, 5 consistency predicates, and 8 technical predicates. Running times for the verification procedure can be found in Table 2.1.

The tight integration into the CAD program and, as such, into the engineer's design process, creates the demand for fast re-evaluation of all conclusions upon small changes to the railway designs.

Usually, engineers start with an empty or draft design and add/change one object at a time. The performance figures presented in Table 2.1 show that the current implementation is well acceptable for "one-shot" validation even for realistic designs with running times in the range of seconds. However, it is not fast enough to *smoothly* and transparently be integrated such that it can automatically rerun the complete verification for each small change.

[4]The Norwegian Railway Authorities (`http://www.banenor.no`).

	Testing station	Arna phase A	Arna phase B
Relevant components	15	152	231
Interlocking routes	2	23	42
Datalog facts	85	8283	9159
Running time (s)	0.1	4.4	9.4

Table 2.1: Case study size and running times on a standard laptop.

An alternative approach that promises to be more efficient is *incremental verification*: instead of solving logic programs from scratch for each verification run, it tries to materialize all consequences of the base facts and then maintains this view under fact updates. Incremental verification is further discussed in Section 2.4 below.

2.4 Incremental verification

While the static infrastructure verification process as developed so far in this text certainly can improve on the current practice of railway signalling design as it is, the full potential of a "light-weight" verification is still unused because of the perceived separation of design activity and verification activity. A verification tool which runs invisibly alongside the design, giving feedback on the current state of the design at any time could have a higher impact on the design process.

The common use case for running the railway design CAD tool in general is that one performs a series of small changes. Indeed, we have found in the collaborations with railway engineers that large portions of the design phase have the goal of *efficiently handling changes* in track layouts, component capabilities, performance requirements, etc. The verification could, instead of being called whenever final version printouts are being made, instantly report potential problems in the design as soon as this information is available.

This requires lowering the running time of the verification, hopefully to less than one second, while keeping in mind that our prototype verification tool should eventually be able to scale up to much larger stations, projects spanning several stations, and significantly larger knowledge bases. Exploiting the fact that the design work is incremental, also evaluating the Datalog programs incrementally seems to be a promising solution to this challenge.

In this section we give an overview of approaches and algorithms for incremental Datalog and the tools that are available. We study these from the viewpoint of our application domain and evaluate initial performance on our case study.

2.4.1 Incremental evaluation of Datalog

Datalog systems use rules to derive a set of consequences (*intensional* facts), from a given set of base facts (*extensional* facts). Typically, Datalog systems use a *bottom-up* (or *forward-chaining*) evaluation strategy, where all possible consequences are materialized [165, Chap. 3] [2, Chap. 13] . This simplifies query answering to simply looking up values in the materialization tables. Any change to the base facts, however, will invalidate the materialization. Several approaches have been suggested to reduce the work required to find a new materialization after changing the base facts.

First, if considering only addition of facts to positive Datalog programs, i.e. without negation, then the standard *semi-naive* algorithm [165, Chap. 3] [2, Chap. 13] is already an efficient approach, as it correctly handles additions to the materialization in an incremental manner. The real challenge is the non-monotonic changes, i.e., when removing facts appearing positively in rules or adding facts appearing negatively in rules. Non-monotonicity is essential in our railway infrastructure verification rules. Graph reachability is prominent in many of the regulations for railway signalling, so efficiently maintaining rules involving transitivity is also essential.

Some algorithms, such as truth maintenance systems [49], work by storing more information (in addition to the logical consequences) about the *supporting facts* for derived facts, so that removal of supporting facts may or may not remove a derived fact, depending on whether the support is still sufficient. This allows efficient removal of facts, at the cost of requiring more time and memory for normal derivations. Inspired by the truth maintenance systems of Doyle [49], the XSB Prolog system implements *incremental tabling* [159] by keeping such sets of supporting facts in memory. Figure 2.6. shows deduced facts for a graph reachability query. In this case, whenever there are several paths connecting a pair vertices of the graph, the `reach` fact for the two vertices is deduced in several ways. In the approach taken in XSB Prolog, different sets of facts that independently prove a derived fact are stored in tables. Whenever changes are made to base facts, the sets of supporting facts can be removed, and as long as the set is not emptied, the derived fact still holds.

```
edge(0,1)
edge(0,2)
edge(1,1)
edge(1,2)
```

(a)

Answer	**Supports**
`reach(0,1)`	⟨1,{`edge(0,1)`}⟩, ⟨2,{`reach(0,1)`,`edge(1,1)`}⟩
`reach(0,2)`	⟨1,{`edge(0,2)`}⟩, ⟨2,{`reach(0,1)`,`edge(1,2)`}⟩

(b)

Figure 2.6: Edge relation and corresponding support sets for a reachability predicate (example from [150]).

Another class of algorithms, working *without* additional "bookkeeping", can be more efficient if the re-evaluation of sets of facts is relatively easy compared to re-materializing all facts. The Propagation-Filtering algorithm [73] works on each removed fact separately, propagating it through to all rules which depend on it, while also after each step of the propagation performing a query for alternative support which would end the propagation. In contrast, the Delete-Rederive (DRed) algorithm [66] is rule-oriented and works on sets of facts, first over-approximating all possible deletions that may result from a change in base facts, then re-deriving any still-supported facts from the over-deleted state before finally continuing semi-naive materialization on newly added facts.

An example where the DRed algorithm is less efficient is graph reachability, which can be encoded on the following form:

$$\begin{aligned}
&\text{path}(x, y) \leftarrow \text{edge}(x, y), \\
&\text{path}(x, y) \leftarrow \text{edge}(x, z) \wedge \text{path}(z, y).
\end{aligned}$$

Figure 2.7 shows key differences in update approaches for the example of a graph reachability from a given node.

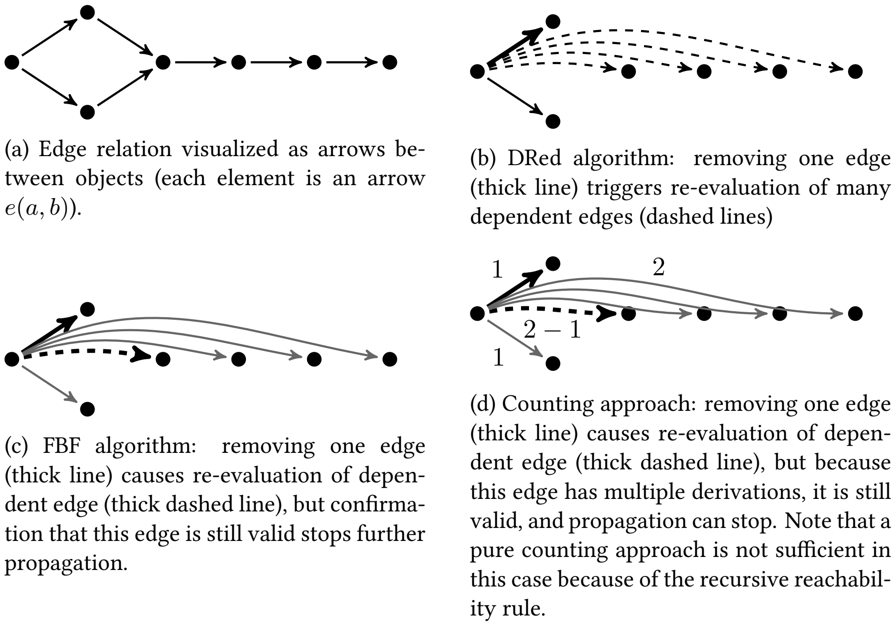
Recently, the Forward/Backward/Forward (FBF) algorithm [118] used in RDFox improved the DRed algorithm in most cases by searching for alternative support (and caching the results) for each potentially deleted fact before proceeding to the next fact. Notably, this method performs better on rules involving *transitivity,* as deletions do not propagate further than necessary.

This method is used in the Semantic Web tool *RDFox*[5], which has a high performance on multicore processors with in-memory databases. We are considering RDFox as an alternative candidates for the back end of our incremental railway infrastructure verification procedure.

2.4.2 Tools and performance

This section summarizes a survey of tools first presented in [108], and describes tools that feature incremental evaluation and Datalog, and which have the maturity required for a future in industrial applications. The logic programs for our verification make use of recursive predicates, stratified negation, and arithmetic. Therefore, we pay particular attention to tools that at least satisfy these needs. In addition, we are looking for high performance on relatively small (in-memory) data sets, so light-weight library-style logic engines are preferred. High-performance distributed "big data" type of tools have less value in this context.

[5]RDFox: scalable in-memory RDF triple store with share memory parallel Datalog reasoning, `http://www.cs.ox.ac.uk/isg/tools/RDFox/`



(a) Edge relation visualized as arrows between objects (each element is an arrow $e(a, b)$).

(b) DRed algorithm: removing one edge (thick line) triggers re-evaluation of many dependent edges (dashed lines)

(c) FBF algorithm: removing one edge (thick line) causes re-evaluation of dependent edge (thick dashed line), but confirmation that this edge is still valid stops further propagation.

(d) Counting approach: removing one edge (thick line) causes re-evaluation of dependent edge (thick dashed line), but because this edge has multiple derivations, it is still valid, and propagation can stop. Note that a pure counting approach is not sufficient in this case because of the recursive reachability rule.

Figure 2.7: Different approaches to incremental evaluation demonstrated on a reachability program using an edge relation. Using the edge relation in (a), the reachability from the first vertex is calculated, and update strategies for (b) DRed, (c) FBF, and (d) a counting approach are exemplified.

XSB Prolog, continuously developed since 1990, has constantly been pushing the state of the art in high-performance Prolog. XSB is especially known for its tabling support [160], which allows fast Datalog-like evaluation of logic programs without restricting ISO Prolog in any way. The tabling support was extended to allow incremental evaluation [150], and these features have been under continued development and seem to have reached a mature state [159]. For some applications, however, the additional memory usage for incremental tabling can lead to a significant increase in the total memory needed.

RDFox is a multicore-scalable in-memory RDF triple store with Datalog reasoning. It reads semantic web formats (RDF/OWL) and stores RDF triples, but also includes a Datalog-like input language which can describe SWRL rules. This rule language has been extended to include stratified negation and arithmetic. The RDFox system also implements the new FBF algorithm for incremental evaluation [118].

RDFox stores internally only triples as in RDF (subject, predicate, and object),

which, in Datalog, corresponds to only using unary and binary predicates. A method of reifying the rules for higher-arity Datalog predicates into binary predicates allows RDFox to calculate any-arity Datalog programs. However, this requires separate rules for each component (argument) of the predicate, and when doing incremental evaluation, the FBF algorithm's backward chaining step then examines all combinations of components (arguments) potentially involved in such a higher-arity predicate. Because of this problem, using RDFox incrementally did not improve running times in our case study, suggesting a need for native support for n-ary predicates in RDFox.

LogicBlox is a programming platform [6] for combining transactions with analytics in enterprise application areas including web-based retail planning and insurance. It uses a typed, Datalog-based custom language LogiQL and has a comprehensive development framework. It claims support for incremental verification, but we could not evaluate it on our railway example due to absence of freely downloadable distributions.

Dyna is a promising new Datalog-like language for modern statistical AI systems [55]. It has currently not matured sufficiently for our application, but its techniques are promising, and we hope to see it more fully developed in the future.

Many other Datalog tools are available (around 30), few of them supporting incremental evaluation. An overview and our brief evaluation of them can be found in the technical report [110], and a more general overview of Datalog tools can be found in the Wikipedia page.[6]

2.4.3 Performance

Table 2.2 compares the running time and memory usage for the verification case study of Arna station presented in Section 2.3, extended to use the incremental capabilities of XSB Prolog. The extra bookkeeping required in XSB to prepare for incremental evaluation requires more time and memory than non-incremental evaluation, so we include both non-incremental and from-scratch incremental evaluation in the table for comparison. We show how updates can be calculated faster than from-scratch evaluation by moving a single object (an axle counter) in and out of a disallowed area near another object (regulations require at least 21.0 m separation between train detectors). Without using abstraction methods, the case study verification uses over 2 GB of memory. So, for any hope of handling larger stations on a standard laptop or workstation, this must be reduced. We were not able to reduce memory usage in this case study using the abstraction methods in XSB (version 3.6.0).

[6] https://en.wikipedia.org/wiki/Datalog#Systems_implementing_Datalog

			Testing station	Arna phase A	Arna phase B
Relevant components			15	152	231
Interlocking routes			2	23	42
Datalog input facts			85	8283	9159
XSB:					
Non-incremental verif.:	Running time:	(s)	0.015	2.31	4.59
	Memory	(MB)	20	104	190
Incremental verif. baseline:	Running time	(s)	0.016	5.87	12.25
	Memory	(MB)	21	1110	2195
Incr. single object update:	Running time	(s)	0.014	0.54	**0.61**
	Memory	(MB)	22	1165	2267

Table 2.2: Case study size and running times on a standard laptop.

While currently none of the tools seem to satisfy all conditions we hoped for in our integration, notably efficiency, but also maturity and stability, it should also be noted that the need for incremental evaluation has been identified by the community not only as theoretically interesting, but also as of practical importance. The RDFox developers aim to support incremental updates of higher-arity predicates in a later version. The XSB project has made efforts to improve its abstraction mechanisms, so future versions might become feasible for our use. If reducing the memory usage would require adapting a Datalog algorithm (such as DRed), then XSB's unrestricted Prolog might be a challenge. A different approach would be to extend another efficient Datalog tool, such as Soufflé[7] to do incremental evaluation, which could require a significant effort.

2.5 Conclusions

We have demonstrated a logical formalism in which railway layout and interlocking constraints can be modelled and technical regulations can be expressed, and which can be decided by logic programming methods (Datalog in particular) with polynomial time complexity. This allows verification of railway signalling designs against infrastructure manager regulations. It also allows to build and maintain a formally expressed body of expert knowledge, which may be exchanged between engineers and automatically checked against designs. We have demonstrated this approach on an ongoing railway design project from the Anacon AS company and

[7]Soufflé: a Datalog compiler, `http://souffle-lang.org/`

using the standard regulations from the Norwegian railway authorities. We have implemented a prototype and integrated it in the engineer's CAD design tool suite. Even though preliminary tests show good performance, we saw the need for faster verification methods, and thus looked into incremental verification tools for Datalog. In this respect we presented our summary of findings and our test results on our railway use case.

2.5.1 Related work

Logic (programming) languages, like Prolog or Datalog, have been used for representing and checking various aspects of railway designs. For the verification of signalling of an interlocking design [88] uses a Prolog database to represent the topology and the layout, where for the the verification, the work uses a separate SAT solver. Similarly, the work of [119, 120] uses logic programming for verification of interlocking systems. In particular, the work uses a specific version of so-called annotated logic, namely annotated logic programs with strong negation (ALPSN). In general and beyond the railway system domain, recent times have seen renewed research interest in Datalog, see for instance the collection [41]. Datalog has in particular been used for formalizing and efficiently implementing program analyses [157, 171], whereas [156] presents Doop, a context-sensitive points-to analysis framework for Java.

The mentioned works generally include *dynamic* aspects of the railway in their checking, like train positions and the interlocking state. This is in contrast to our work, which focuses on checking against a formalization of the general design rules issued by the regulatory bodies, thus concentrating on static aspects such as the signalling layout. This makes the notorious state-space explosion problem less urgent and makes an integration into the standard design workflow within the existing CAD tool practical.

Lodemann et al. [103] use semantic technologies to automate railway infrastructure verification. Their scope is still wider than the approach described in this chapter, in the computational sense, with the full expressive power of OWL ontologies, running times on the order of hours, and the use of separate interactive graphical user interfaces rather than integration with design tools.

2.5.2 Future work

In the future work with Railcomplete AS, we will focus on extending the rule base to contain more relevant signalling and interlocking regulations, and also on evaluating the performance of our verification on a larger scale. Design information and rules about other railway control systems, such as geographical interlockings and train protection systems could also be included. The current work is assuming Norwegian regulations, but the European Rail Traffic Management System is expected to dominate in the future.

Involving railway engineers in knowledge base development is somewhat hindered by the fact that Datalog and logic programming, though declarative and concise, are still programming languages, and a good intuition about language semantics is required for efficient and correct development. One possible mitigation for this using Controlled Natural Language as a front-end for inputting verification properties is presented in Chapter 5

Finally, the Datalog language is not well suited to model dynamic analyses involving trains moving on the track and how the signalling design impacts capacity and capabilities. This question is treated in the two following chapters.

3 Dynamic analysis using local capacity specifications

Railway capacity is complex to define and analyse, and existing tools and methods used in practice require comprehensive models of the railway network and its timetables. Design engineers working within the limited scope of construction projects report that only ad-hoc, experience-based methods of capacity analysis are available to them. Designs have subtle capacity pitfalls which are discovered too late, only when network-wide timetables are made – there is a mismatch between the scope of construction projects and the scope of capacity analysis, as currently practised.

In this chapter, we suggest a language for capacity specifications suited for construction projects, expressing properties such as running time, train frequency, overtaking and crossing. Such specifications can be used as contracts in the interface between construction projects and network-wide capacity analysis.

We show how these properties can be verified fully automatically by building a special-purpose solver which splits the problem into two: an abstracted SAT-based dispatch planning, and a continuous-domain dynamics and timing constraints evaluated using discrete event simulation. The two components communicate in a CEGAR-loop (counterexample-guided abstraction refinement). This architecture is beneficial because it clearly distinguishes the combinatorial choices from continuous-domain calculations, so that the simulation can be extended by relevant details as needed. We describe how loops in the infrastructure can be handled to eliminate repeating dispatch plans, and use case studies based on data from existing infrastructure and ongoing construction projects to show that our method is fast enough at relevant scales to provide agile verification in a design setting.

3.1 Capacity in railway construction projects

The planning and engineering of a railway control system has safety as primary requirement. Secondary to safety, the notion of performance and capacity of a railway control system remains more elusive. The capacity of a railway control system, and thus of railway infrastructure in general, is hard to define precisely (see [70, 3, 97]). Any capacity measure will necessarily make assumptions about the operation of the railway. One can say that the railway infrastructure does not have an inherent capacity, only capacity for specific use cases. A fully accurate assessment of capacity can only be made under a fully specified timetable, meaning that every train's arrival and departure times at all stations in the net-

work must be known. This makes for a highly coupled analysis, as constructing an actual timetable requires bringing together details about infrastructure, rolling stock, transportation demands, and crew schedules. Systematic capacity analysis for railways is typically performed on the scale of national railway networks, using comprehensive input on infrastructure and timetables, and only after planning and engineering has produced a final design. Moreover, the widely used methods and tools for capacity analysis are heavy-duty methods, consisting of complicated simulations, and require specialized knowledge, thus not being suitable for more agile design-time verification of railway stations.

For construction projects and control system engineering, it would not be feasible to use a fully specified timetable for verifying that the control system will be able to provide the required capacity, because (1) detailed timetabling and capacity analysis takes too much effort and specialized knowledge, and is usually saved for later stages of design, and (2) the design of a control system cannot or should not depend too heavily on other parts of the network, as these parts may also change in the future.

Another approach to capacity analysis is the so-called analytical capacity approach, which views the railway network as a network of queues, or a maximum flow problem, abstracting away the low-level discrete behaviour while preserving the high-level continuous behaviour. These methods can give preliminary or low-precision network-wide results, but fail to account for the critical factors which arise when performance is pushed to the limit. Simplifying assumptions that can be suitable for network-scale capacity analysis, such as instantaneous speed changes, or fixed travelling times between different locations, are usually not suitable for infrastructure design. Specifically, disregarding the discrete allocation logic of the interlocking system, and the position and velocities of individual trains, makes these methods unsuitable for analysis of signalling design. The detailed optimization of signal and detector locations needs to account for a detailed model of train dynamics and control system behaviour exactly because higher-level analysis requires this assumption of local optimization to the simplified behaviours used in network-global analysis.

As none of these techniques are particularly well-suited, railway engineers working on construction projects usually rely on informal, vague, or even non-existent capacity specifications, and need to make ad-hoc analyses of how the control system might provide this capacity.

In consequence, we address the following problem: in the context of designing the layout and control systems for railway stations, does the station infrastructure have the *capacity* to handle the amount of trains and the desired travelling times to provide adequate service in transportation of goods and passengers?

As an example, consider the question of crossing trains on a railway station. Figure 3.1 shows two sequences of movements which result in such a crossing. There are a number of details of the railway design which can cause this scenario

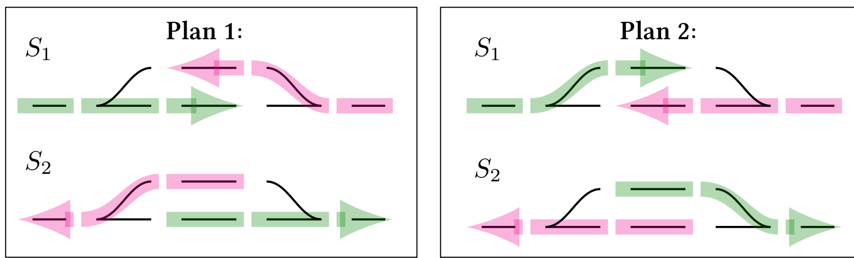


Figure 3.1: Two alternative plans for achieving a crossing of two trains on a two-track station. The green areas show track segments which are currently occupied by a train going from left to right, while the pink areas show track segments which are currently occupied by a train going from right to left.

to become infeasible (or take an unacceptably long time), such as signal placement, detector placement, correct allocation and freeing of resources, track lengths, train lengths, etc.

Railway design and construction planning is an old engineering discipline with long-standing traditions. Demands for the highest safety, compatibility with existing infrastructure and practices, and high investment costs, make railway engineering a conservative domain. The design process of railways is in practice highly sequential, leading to the well-known effects of so-called waterfall process models.

Waterfall-style design processes require that high-level specifications can be written up-front and afterwards implemented without feedback from the implementation process back to the high-level specifications. This also means that verification and validation in waterfall-style design processes is confined to the scope of each separate design activity, or destined to have little hope of improving the design when weaknesses are uncovered.

Unfounded design assumptions made early in the early process stages have been known to trickle all the way down to the final stages and require new rounds of design starting from the top, a process which can take several years.

These negative effects are typically mitigated by:

1. Re-using proven design concepts, i.e. doing something the same way as somewhere else, where it has already turned out to work well.

2. Allowing sizeable margins, e.g. planning the track with more than enough space for safety distances so that it is highly likely that control system engineers will later be able to come up with a safe and performant design.

These mitigations exploit tradition, experience and cross-discipline knowledge in the railway engineers, which in turn contributes to making the engineering community slow-moving and conservative.

However, modern construction practice expects and demands optimization. When space requirements, performance requirements and cost limitations are squeezed to the limits of the possible, the tradition-based railway engineering approach lacks the methods to accurately reason about the limitations of the finished system from partially finished design plans.

Using *agile verification* of high-level properties from the beginning of a design project, and in every step of the process, allows engineers to better see the consequence of each decision, and immediately uncover errors and shortcomings that would otherwise be discovered only months or years later.

Our goal is to develop a verification technique and tool to help engineers specify capacity properties *at design time* and to check these automatically. To be agile, the tool needs to (1) have reasonable running times so that the verification can be run on the fly as the design is being updated by an engineer working in a drafting CAD application, and (2) keep the required input to the minimum of information needed to verify relevant properties. This style of verification gives engineers immediate feedback on their design decisions while requiring small amounts of specification and verification work.

3.1.1 Problem definition

We consider **the low-level railway infrastructure capacity verification problem**, which we define as follows:

> Given a railway station track plan including signalling components, rolling stock dynamic characteristics, and a performance/capacity specification, verify whether the specification can be satisfied and find a dispatch plan as a witness to prove it.

This problem concerns the following railway infrastructure design activities:

- Low-level **running time** analysis – verify the time required for getting from point A to point B.
- Low-level **schedulability** analysis – verify frequency of trains arriving at a station, and simultaneous opportunities for crossing, parking, loading, etc.
- **Combinations** – verify running time requirements on schedulable operations.

3.1.2 Approach

To work with capacity in a way that is suitable for construction projects, we suggest a formalization of capacity requirements as a set of operational scenarios involving a set of trains, a set of locations to visit, and a set of timing constraints.

Verification in this domain can in principle be encoded into the SMT [10, 42, 123] or PDDL+ [59] languages, essentially resulting in a SAT modulo non-linear

real arithmetic problem [60, 84]. Many solvers can handle such problems [43, 62, 51], but we found that the problem size of our test cases, in terms of the number of planned actions and in terms of number of interacting Boolean and non-linear real logic terms, were out of reach for agile verification. Also, train dynamics using only constant acceleration $x'' = c$ is in some cases too simplistic for engineering. We would like to be able to extend the dynamics equations using e.g. polynomials of higher order or even numerical integration.

Therefore, we have developed a verification tool chain that uses a simple CEGAR loop [34] between a SAT-based planning tool that works on a discrete abstraction of control system commands, and a discrete event simulation engine (DES) [147] that calculates detailed continuous results for a specific plan, taking the physics of moving trains into account.

The SAT-based planner uses bounded model checking (BMC) [12] where time is reduced to a series of partially ordered actions with unknown durations, and the choice of actions are the available commands in the control system. The DES component verifies the continuous time/space results given the Boolean decisions of control system commands, and adds new SAT constraints excluding unsatisfactory solutions.

The separation of discrete and continuous domains also has the advantage that the simulation component can be extended to handle more complex models, such as engine power curves, tunnel air resistance, curve rolling resistance, train weight distribution, etc., without affecting the planning logic or its computational complexity.

We have tested our method and tool on practical examples from existing infrastructure and ongoing construction projects in collaboration with railway engineers in Railcomplete AS.

3.2 Dynamic behaviour

To verify performance properties, we need to find a sequence of trains and elementary routes for the train dispatcher, i.e., a *dispatch plan*, which when executed under the safety and correctness constraints decribed below, demonstrate the properties described in the performance requirements (detailed in Sec. 3.3).

Low-level analysis of train movements covers a wide range of constraints given by the track layout, the control system, and operational procedures, to be certain that the analysis produces detailed, realistic results. The following subsections give an overview of these constraints, divided into four classes. See [131] for a more in-depth description of railway operation principles.

3.2.1 Physical infrastructure

Trains travel on a network of railway tracks which have physical properties such as length, gradient, curvature, etc. Tracks branch off using *switches*, whose *setting* determines where the train goes. Detectors on the track are used by the control system to determine whether track segments are occupied. The physical infrastructure also determines the *sight areas*: the set of locations where a train receives information from a given signal.

3.2.2 Interlocking: allocation of resources

The safety-critical control systems for railway infrastructure are called *interlockings*. An interlocking takes requests for activating routes from a dispatcher. When a route is activated, switches are moved into correct positions and signals are set to show the go aspect. The interlocking is also responsible for assuring that activating the route, i.e. allowing the train to travel the route, is safe. This safety is ensured through the following requirements:

- Routes require the exclusive allocation of **track segments**, so that two routes which use some of the same track segments cannot be activated at the same time. Routes must be allocated as a unit, i.e. all segments must be free at the time of allocation. However, track segments may be de-allocated to other routes as soon as the train has passed a segment.

- **Switches** need to be in the correct position for the train to travel along the route. Also, the switches must be locked, so that they cannot accidentally be moved while the train is travelling, and detectors on the switch must report that the switch is in the correct position, and correctly locked.

- A **safety zone** (also called overlap) beyond the end of the route must be vacant, but not necessarily exclusively allocated, i.e. two safety zones may share track segments. The safety zone is released after a given time which is long enough that it is unlikely that the train is still moving forward. This timeout is calculated based on the length of the route.

- Routes which pass through switches require specific track elements to cover any potential movements into the route path. This is known as **flank protection**, and cover can typically be provided by signals, switches or other objects.

- **Signals** can only show the go aspect when it is the starting point for a currently active route, in all other states, the signal must show the stop aspect. Distant signals, i.e. additional signals showing information about the next upcoming route, must give information consistent with the upcoming signal.

These constraints are explicitly expressed for a given railway station through the interlocking specification, which is an artifact of the design process.

Avoiding collisions by exclusive use of resources is the responsibility of the interlocking, which takes requests from the dispatcher for activating **elementary routes**. An elementary route is the smallest unit of resources that can be allocated to a train, see Figure 3.3. Route activation is a process which proceeds as follows:

1. Wait for all **required resources**, such as track segments and switches, to be free. Resources required by a route are typically any resource in the train path (or sometimes outside of it), which ensure that all movements are performed at a safe distance from each other.

2. **Movable elements** (e.g. switches) must be set to correct positions. If they are not, start a sub-process which moves the element into place, and wait for this process to finish before proceeding.

3. **Signals** are then set to show the 'proceed' aspect to the train when the above steps are finished. When the front of the train has passed the signal, it is immediately reset to show the 'stop' aspect.

4. A **release** process is started, which waits for the train to finish using the allocated resources (i.e. to travel over them) and frees them when this has happened.

3.2.2.1 Influence of safety zones on capacity

The safety zone, as described above, is a set of track sections and switches allocated together with a route to ensure that slightly overrunning a signal showing the stop aspect is safe. Different manufacturers and national regulations have various ways of specifying how a safety zone is released and how alternative safety zones are implemented. The main variations are:

1. The safety zone from a route end point persists until a route is allocated from the end point. This can be problematic if the safety zone blocks other traffic or if the train is changing directions and not proceeding past the end point. The following two methods are the usual mitigations for these problems.

2. The safety zone is released after a pre-set time. This time should be long enough so that the probablilty that the train is still running towards the end point is very low.

3. The safety zone it not released, but can be replaced by another safety zone from the same route end point. This method is called *swinging overlap* in British English.

See Figure 3.2.

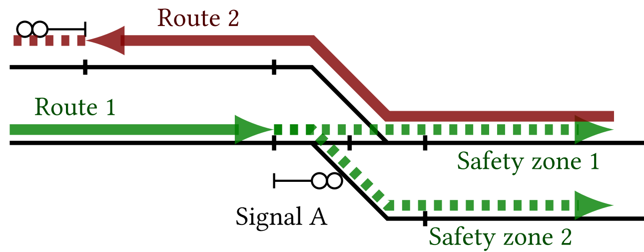


Figure 3.2: An elementary route 1 ending in signal A can protect trains from overrunning the signal by allocating one of the safety zones (shown as safety zone 1 and 2). In some situations, safety zone 1 might be preferred so that the switch following signal A is in the correct position for letting the train in route 1 proceed quickly. However, allocating safety zone 1 blocks route 2 from use. So in other situations, safety zone 2 might be preferred, for example for two trains to concurrently enter a station. Some control systems may allow one safety zone to be replaced by another after the route has been allocated.

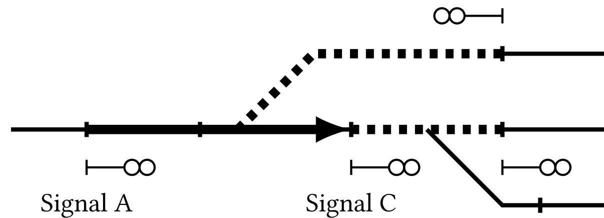


Figure 3.3: Elementary route AC from signal A to the adjacent signal C. The thick line indicates track segments on the train's path which are reserved for this movement, and the dashed lines indicate reserved track segments outside the path.

3.2.3 Communication constraints

After movement has been allowed by the control system, the driver must be informed of this fact. When a route is activated, a train inside the sight area of the route's entry signal reads the signal's message that movement authority is given. The train driver may then drive the train forward until the next signal. The following types of signalling systems are common in railways:

- Traditional signaling with trackside lamps. Communication is limited by how many different aspects the lamps can show. To avoid high-speed trains slowing down at every signal, several consecutive elementary routes can be signaled in advance using so-called distant signals.

- Automatic train protection systems (ATP) work similarly to signals, but may give more information. Many ATP systems communicate information through magnets or short-range radio at specific locations on the track, corresponding to a signal sight area of zero length.
- The European Rail Traffic Management System (ERTMS) currently being implemented in many European countries replaces lamp signals with trackside marker boards, and uses long-range radio for communication. This effectively removes the communication constraint, as the radio can be used to update any train's movement authority at any time.

To avoid trains slowing down or stopping at every signal, several consecutive routes can be allocated and signaled in advance using so-called distant signals. The amount of information that can be transmitted to the train drivers through the signaling puts a constraint on how far ahead the routes can be pre-allocated.

- Traditional signaling with track-side lamps are limited by how many different aspects the lamps can show. Signals can typically show information about either one or two routes, but some countries have extended to information about three consecutive routes. It is also common to extend the information given by signals using track-side electronic communication.
- In the version of ERTMS called *Level 2*, which is currently the most used system, lamp signals are replaced by trackside marker boards, and the actual communication goes over radio. This effectively removes the communication constraint, as the radio can be used to communicate any movement authority at any time.

3.2.4 Laws of motion

Trains move within the limits of given maximum acceleration and braking power, so train drivers need to plan ahead for braking so that the train respects its given movement authority and speed restrictions at all times.

The speed increase from v_0 to v over a time interval Δt is limited by the train's maximum acceleration a:

$$v - v_0 \leq a\Delta t.$$

However, when there is a more restrictive speed restriction ahead, the driver must start braking in time to meet the restriction. A signal showing the 'stop' aspect can be treated as a speed restriction of zero. Since speed restrictions change with time, the driver must re-evaluate their actions whenever new information is received.

A train has the following constraint on its velocity v for each restriction,

$$v^2 - v_i^2 \leq 2bs_i,$$

where v_i is the maximum allowed speed, s_i is the distance to the location where the restriction starts, and b is the maximum retardation achieved by braking. These restrictions are given as (1) constant maximum velocity restrictions given by signs beside the track, or (2) dynamical velocity restriction given by the distance to the next stop signal (i.e. the length of the movement authority).

3.3 Performance requirement properties

To capture typical performance and capacity requirements in construction projects, we define an **operational scenario** $S = (V, M, C)$ as follows:

1. A set of **vehicle types** V, each defined by a length l, a maximum velocity v_{max}, a maximum acceleration a, and a maximum braking retardation b.

2. A set of **movements** M, each defined by a vehicle type and an ordered sequence of visits. Each visit q is a set of alternative locations $\{l_i\}$ and an optional minimum dwelling time t_d.

3. A set of **timing constraints** C, each constraint consisting of two visits q_a, q_b, and an optional numerical constraint t_c on the minimum time between visit q_a and q_b. The two visits can come from different movements. If the time constraint t_c is omitted, the visits are only required to be ordered, so that $t_{q_a} < t_{q_b}$.

To demonstrate how an operational scenario captures requirements of railway construction projects, we give some examples using the syntax of the file format used in our tool[1]. First, we define the following vehicle types:

```
vehicle passengertrain length 220.0
  accel 1.0 brake 0.9 maxspeed 55.0
vehicle goodstrain length 850.0
  accel 0.5 brake 0.5 maxspeed 20.0
```

The following set of **performance specifications** are selected prototypical versions of specifications that railway engineers have suggested as useful for automated verification:

- **Running time**: expresses an expectation of how long it should take for a train to travel between two locations. To specify this, we simply require that a train visits some location `b1` and later visits some other location `b2`. A timing constraint of 90.0s between these visits sets the running time requirement.

[1] For details of the input file formats, see `https://luteberget.github.io/rollingdocs/usage.html`

```
movement passengertrain {
  visit #a [b1]; visit #b [b2] }
timing a <90.0 b
```

- **Train frequency**: a train station processes a set of trains arriving and departing with a fixed frequency. On a two-track station, we exemplify a sequence of four trains and their relative departure times.

```
movement passengertrain {
  visit [b1]
  visit [platform1,platform2] wait 60.0
  visit #e1 [b2] }
// ...3 more trains with visits e2, e3, e4.
timing e1 <90.0 e2
timing e2 <90.0 e3
timing e3 <90.0 e4
```

- **Overtaking**: trains travelling in the same direction can be reordered. For example, we specify a passenger train travelling from `b1` to `b2`, and a goods train with the same visits. Timing constraints ensure that the passenger train enters first while the goods train exits first.

```
movement passengertrain {
  visit #p_in [b1]; visit #p_out [b2] }
movement goodstrain {
  visit #g_in [b1]; visit #g_out [b2] }
timing p_in < g_in
timing g_out < p_out
```

- **Crossing**: trains travelling in *opposite directions* can visit this station simultaneously. This example is similar to the previous one, but the goods train now travels in the opposite direction, and the timing constraints require that the trains are inside the model simultaneously.

```
movement passengertrain {
  visit #p_in [b1]; visit #p_out [b2] }
movement goodstrain {
  visit #g_in [b2]; visit #g_out [b1] }
timing p_in < g_out
timing g_in < p_out
```

Similar specifications, and combinations of such specifications, are relevant in most railway construction projects. Since we typically only need to refer to lo-

cations such as model boundaries and loading/unloading locations, these specifications are not tied to a specific design, and can often be re-used even when the design of the station changes drastically.

Stations can be designed to be either intermediate stops, or end-of-line stops. Also, some stations are intermediate stops for some trains while also being the end-of-line stop for other trains. A challenge of planning for end-of-line stops is that the train must usually be allowed to turn around and go back in the direction it came from. Allowing an unbounded number of such turns can in principle lead to an infinite number of dispatch plans.

The turning is also related to the challenge of having loops in the infrastructure. This is uncommon within a single station, but can sometimes occur in construction projects where several stations together form a loop topology. Also here, we must take care not to explore or suggest an infinite number of ways to execute an operational scenario. These aspects are treated in Section 3.4.2.

3.4 Tool chain and solver architecture

Being able to do performance verification (i.e., capacity analysis) automatically, using only information typically available in a construction project, is potentially a valuable tool for railway engineering in the disciplines of signalling and interlocking. In this section, we describe our approach to verifying capacity properties using the formalization of capacity given in the previous section.

We have investigated several logic-based approaches for the domain and problem described above. The PDDL+ language has been designed to express planning problems in mixed discrete/continuous domains. As each discrete change is represented by a planning step, our test case problem instances would need at least 50-100 steps to be solvable. We were only able to solve the most trivial test cases in less than one second using the SMTPlan+ solver.

Encoding into SMT can be done by expressing planning as a BMC problem. This approach suffers from the same problem of having a high number of planning steps (some improvements can be made, s.a. making train driver choices implicit in constraints on the relation between velocity, distance and time).

To address these limitations, we developed a CEGAR-style tool which exploits the limited number of control system commands to make an abstraction of the planning problem, see Figure 3.4.

A verification tool chain which solves the low-level railway infrastructure capacity verification problem and supports agile verification in railway construction projects is outlined in Figure 3.5. The manual, source code and test cases are available online[2]. The tool uses the MiniSat v2.2.0 solver.

[2]`https://luteberget.github.io/rollingdocs` and `https://github.com/koengit/trainspotting`

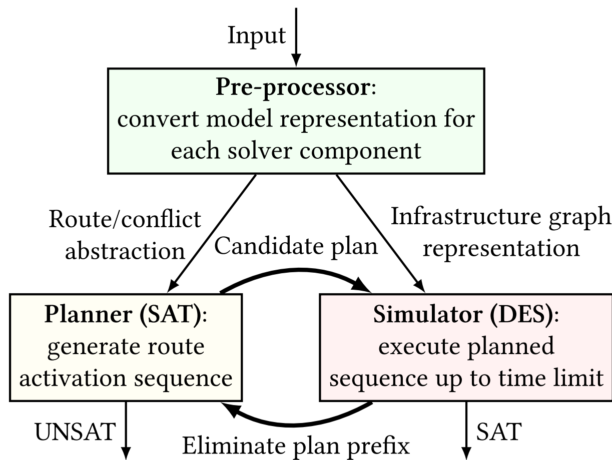


Figure 3.4: Conceptual diagram of CEGAR architecture. Infrastructure, routes, train types, and movement specifications are transformed into (1) the planner's abstract representation, containing only elementary routes and train lengths, and (2) the detailed graph representation used in the simulator component.

The tool is complementary to other verification techniques in railway design, such as static layout verification [109, 107, 105], static interlocking verification [75, 107], interlocking program verification [20], and timetable analysis [74].

The following input documents are used:

- **Operational scenarios** defining the performance properties to verify. Examples are given in Section 3.3.
- **Infrastructure** given in the railML format [121, 139]. In our case studies, we exported railML files using the RailCOMPLETE software, a plugin for the widely used AutoCAD drafting software. Using a model taken directly from the drafting program means that no additional model preparation is needed.
- **Elementary routes** (*optional*), given in a custom format which is compatible with the upcoming railML interlocking format. Although subject to design, a decent guess of the content can be straight-forwardly derived from the infrastructure by listing resources on paths between adjacent signals, so this input is optional.
- **Dispatch plans** (*optional*) corresponding to each operational scenario. The verification tool can produce dispatch plans fulfilling the performance specification, so this input is optional.

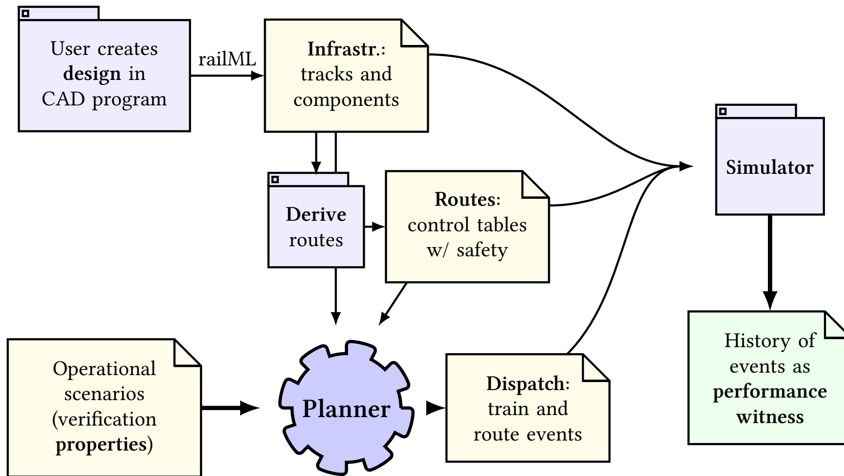


Figure 3.5: Capacity verification tool chain overview. Yellow boxes represent input documents. Note that only infrastructure and operational scenarios are strictly required – interlocking tables can be derived, and dispatch plans can be synthesized. Blue boxes represent programs. The green box represents the output document from the simulator, which is a history of events which is the witness that proves the performance requirement.

An advantage of the separation of planner and simulator is that each component can be used separately. *The planner alone* may be used to enumerate different possibilities for train movements, which might be used in an operational testing situation. *The simulator alone* may be used to debug the execution of a specific dispatch plan to examine performance deficiencies, and educationally for demonstrating the workings of the railway system. Put together, the two components provide automated verification, which is the main goal of our efforts. It would also, in principle, be possible to use one of the commercial simulation packages, such as OpenTrack or RailSys, provided that all input and simulation control can be given though a programmable interface (API).

3.4.1 Dispatch Planning using SAT

The planner solves the abstracted discrete planning problem of finding a dispatch plan, i.e., determining a sequence of trains and elementary routes which make the trains end up visiting locations according to the movements specification.

We encode an instance of the abstracted planning problem into an instance of the Boolean satisfiability problem (SAT). We consider the problem a model checking problem, and use the technique of bounded model checking (BMC) to unroll

the transition relation of the system for a number of k steps, expressing state and transitions in propositional logic.

Using BMC for planning works by asserting the existence of a plan, so that when the corresponding SAT instance is satisfiable, it proves the fulfillment of the performance requirements and gives an example plan for it. When unsatisfiable, we are ensured that there is no plan within the number of steps k. In practice plans with higher number of steps are not of interest; i.e., the bound k is chosen based on practical considerations (e.g., twice the number of trains was sufficient in our case studies). The SAT instance is built incrementally by solving with $k-1$ steps and then adding the kth step if necessary.

The abstracted planning problem is encoded as a SAT instance by representing states, constraints on each state, and constraints on consecutive states. *State i* of the system in the planner component is represented as:

1. Each route r_j has an **occupancy status** $o^i_{r_j}$ which is either free ($o^i_{r_j} = \text{Free}$) or occupied by a specific train t_k ($o^j_{r_i} = t_k$). Each combination of route and train is represented by a Boolean variable, but we will write constraints with $o^i_{r_j}$ as a variable from the set of trains.

2. Each route also has a choice from its associated **safety zones** $z^j_{r_j} \in \{1 \ldots n\}$ which determines which other routes are considered to be in conflict (see Figure 3.6).

3. Each train t_k has a Boolean representing **appearance status** b^i_k, used to propagate to future states that a train has started (used in constraint C2 below).

4. Each visit l has a Boolean representing **required visits** v^i_l, which is used to propagate to future states that a visit requirement has been fulfilled (used in constraint C5).

5. Each combination of route r_j and train t_k has a Boolean representing **deferred progress** $p^i_{j,k}$, used to propagate to future states that a train is not progressing, and must resolve the conflict in the future (used in constraint C8).

A dispatch plan is produced directly from the occupancy status $o^i_{r_j}$ and safety zone choices $z^j_{r_j}$ of states by taking the difference between consecutive states and then dispatching any trains and routes which become active from one state to the next. If swinging safety zones (also known as *swinging overlaps*) or safety zone timeouts are enabled, then consecutive steps can have different safety zones, and when this happens, a swing command is also added to the dispatch plan.

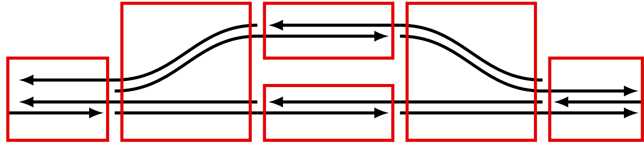


Figure 3.6: The planner component takes an abstracted view of the railway infrastructure. Lines represent elementary routes with traveling direction given by the arrows. Boxes indicate routes in conflict, i.e. only one of them can be in use at a time.

Constraints are applied to each state and each pair of consecutive states to ensure that:

- The plan is viable for execution (i.e., **correctness**):

(C1) Conflicting routes are not activated simultaneously.

(C2) Each train can only take one contiguous path.

(C3) An elementary route must be allocated as a unit, but its parts may be deallocated separately.

(C4) (Partial) routes are deallocated only after a train has fully passed over them.

- The **plan fulfills capacity specifications**:

(C5) Trains perform their specified visits.

(C6) Visits happen in specified order.

- Equivalent solutions are eliminated (for **performance**):

(C7) Routes are deallocated immediately after the train has fully passed over them.

(C8) A train's path is extended as far as possible in the current time step, unless hindered by a conflicting train (i.e., **maximal progress**).

Equivalent plans, which result in the same trains traversing the same paths and conflicting in the same locations, should have the same representation so that enumeration of different plans produces meaningful alternatives. For example, the two dispatch plans for crossing shown in Figure 3.1 are the only two alternatives given by the planner for this operational scenario. See Figure 3.7 for other dispatch plans which fulfil the correctness constraints (C1-6) but which do not have maximal progress in each state.

The simulator component, which evaluates the time consumption of plans, reports which parts of the plan fail the timing constraints, and the negation of this

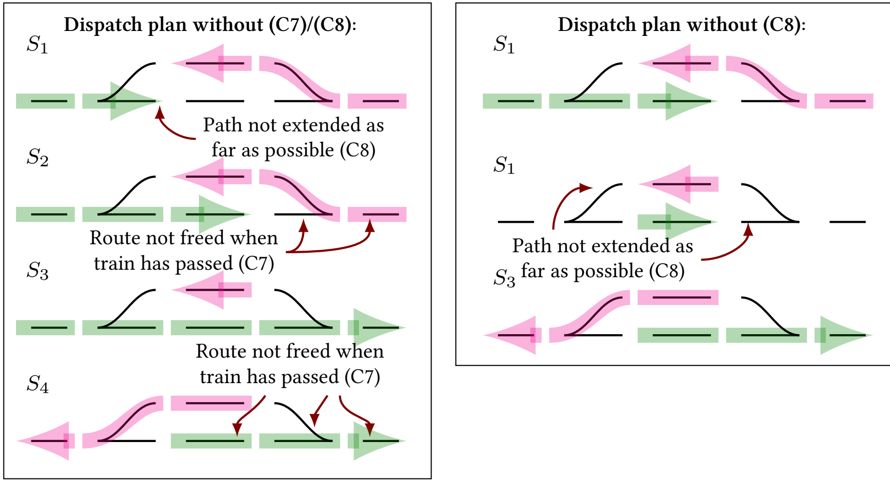


Figure 3.7: Examples of dispatch plans which are correct plans (constraints (C1-6)), but which have better equivalent descriptions that allocate and deallocate as soon as possible. These plans do not fulfil constraints (C7) and (C8). Compare with plan 1 in Figure 3.1.

partial plan is added to the SAT instance. Since the timing calculations are path dependent, we use the part of the plan starting from the beginning and going up to the step where the timing specification violation occurs. This way of refining the abstraction can cause performance problems when many different choices are possible early in the plan, and the timing violation can only be found near the end of the plan, as demonstrated in Section 3.6. Finding a way to make more precise refinements could be necessary for larger problem instances.

The implementation of each of these constraints as propositional logic statements is described below. Constraints apply separately to all states i unless noted otherwise.

3.4.1.1 Resource conflicts (C1)

Any two routes which require the same resources cannot both be allocated in the same state.

$$\forall r_a \in \text{Routes} : \forall r_b \in \text{conflict}(r_a) : o^i_{r_a} = \text{Free} \lor o^i_{r_b} = \text{Free}.$$

3.4.1.2 Train path (C2)

At most one alternative route is taken by a train in a single state. First, ensure that only one route from a given start signal may be taken at any time.

$$\forall t \in \text{Trains} : \forall s \in \text{Signal} : \text{atMostOne}(\{o_r^i = t \mid \text{entry}(r) = s\}) \ .$$

We use a standard sequential encoding to encode atMostOne and other similar constraints, as explained in e.g. [155]. Note that entry signals for all routes entering from a model boundary share the same null value, so that this constraint also excludes plans where a single train appears in several positions at once. Each train should only enter the plan once, thus the appearance Boolean changes to true in exactly one transition.

$$\forall t \in \text{Trains} : b_t^i \Rightarrow b_t^{i+1}.$$

$$\forall t \in \text{Trains} : \text{exactlyOne}\left(\left\{\neg b_t^j \wedge b_t^{j+1} \mid j \in \text{States}\right\}\right) .$$

A train appears when an entry boundary route is allocated:

$$\forall t \in \text{Trains} : \forall r \in \{r \in \text{Routes} \mid \text{entry}(r) = \text{null}\} : \left(o_r^i \neq t \wedge o_r^{i+1} = t\right) \Rightarrow b_t^{i+1}.$$

Routes which are not entry routes can only be allocated to a train when they extend some other route which was already allocated to the same train, i.e., consecutive routes must match so that the exit signal of one is the entry signal of the next:

$$\forall t \in \text{Trains} : \forall r \in \{r \in \text{Routes} \mid \text{entry}(r) \neq \text{null}\} :$$
$$\left(o_r^i \neq t \wedge o_r^{i+1} = t\right) \Rightarrow \bigvee \left\{o_{r_x}^{i+1} = t \mid r_x \in \text{Routes},\ \text{entry}(r) = \text{exit}(r_x)\right\} \ .$$

Note that this constraint ensures that the trains' allocation to routes *locally* forms a path in the graph of routes. In the presence of cycles, this constraint does not rule out cyclic allocations disjoint from the rest of the train's path. This problem is handled separately in Section 3.4.2 below.

3.4.1.3 Partial release (C3)

Partial release is represented by splitting each elementary route into separate routes for each component which is released separately. The set *Partial* contains such sets of routes. Partial routes are allocated together (see Figure 3.8):

$$\forall t \in \text{Trains} : \forall q \in \text{Partial} : \text{allEqual}(\{o_r^i \neq t \wedge o_r^{i+1} = t \mid r \in q\})$$

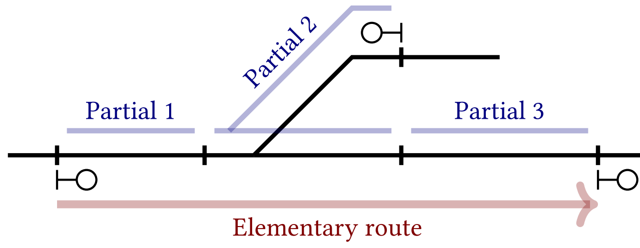


Figure 3.8: The planning abstraction of the train dispatch allocates a set of partial routes to each train. Elementary routes are sets of partial routes which must always be allocated together.

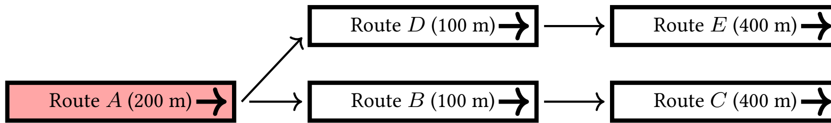


Figure 3.9: When a train of length 200.0 m has been allocated to route A, that route can only be freed when the train has been allocated to either both B and C or both D and E.

3.4.1.4 Deallocation (C4, C7)

Routes are freed when sufficient length has been allocated ahead to fully contain the train.

$$\begin{aligned}&\forall t \in \text{Trains} : \forall r \in \text{Routes} :\\&o_r^i = t \Rightarrow \left((o_r^{i+1} = t) \Leftrightarrow \text{freeable}_{r,t}(\{o^i\})\right) \ .\end{aligned}$$

Note that the equality sign on the right hand side implies that deallocation is both allowed (C4) and required (C7). The freeable predicate is a disjunction of paths (conjunction of routes) ahead which are long enough to contain the train. For example, on the routes shown in Figure 3.9, if route A holds a train t of length 200.0 m, freeing A is constrained by:

$$A^i \Rightarrow \left(A^{i+1} \vee (B^i \wedge C^i) \vee (D^i \wedge E^i)\right) .$$

3.4.1.5 Visits (C5, C6)

Visits and their order are given by the set VisitOrder, which contains pairs of (t, v), where t is a train and v is a set of alternative routes. Visits must happen using any

of the alternative routes, and must be in an order such that the visit (t_1, v_1) comes before (t_2, v_2):

$$\forall((t_1, v_1), (t_2, v_2)) \in \text{VisitOrder} :$$
$$\bigvee \left\{ o^i_{r_a} = t_1 \wedge o^j_{r_b} = t_2 \wedge i \leq j \;\mid r_a \in (v_1), r_b \in (v_2), i, j \in \text{States} \right\} .$$

3.4.1.6 Forced progress (C8)

In addition to the constraints on allocation and freeing required to produce a valid plan, we also add constraints which force each train to get allocated routes further along a path forward unless there is a conflict. Routes ahead are either allocated, or the train is deferred p:

$$\forall t \in \text{Trains} : \forall r \in \text{Routes} : o^i_r \Rightarrow p^i_{t,r} \vee \bigvee \left\{ o^i_{r_x} \mid r_x \in \text{Routes},\ \text{entry}(r_x) = \text{exit}(r) \right\}$$

Deferred progress must be resolved by freeing a conflicting route, and then allocating it to the train in the following step:

$$\forall t \in \text{Trains} : \forall r \in \text{Routes} :$$
$$p^i_{t,r} \Rightarrow p^{i+1}_{t,r} \vee \bigvee \{ o^i_{r_c} \neq \text{Free} \wedge o^i_{r_x} \neq t \wedge o^{i+1}_{r_x} = t$$
$$\mid r_c, r_x \in \text{Routes},\ \text{exit}(r) = \text{entry}(r_x),\ r_c \in \text{conflict}(r) \}$$

When i is the last state, $p^{i+1}_{t,r}$ is considered to be false, which forces the deferred progress to be resolved eventually. Note that it is not required that the conflicting trains are distinct.

3.4.2 Handling turning and loops

Many railway construction projects have only *acyclic* infrastructure, in the sense that trains enter from one side of the station and exit on the other side, and all paths from one side to the other are acyclic. However, if the infrastructure has a same-directed cycle which can be allocated without conflicting with other routes, the constraints C2 above are insufficient to ensure train path consistency, see Figure 3.10. The train path consistency constraints described in the previous section require each active route to have a route before it already being active. This works in the acyclic case, because the chain of routes always leads back to either a model boundary or a route already allocated in the previous step. With cyclic infrastructure, however, a sequence of routes can justify each other, which would lead to a train appearing out of nowhere. It is a known problem that expressing this kind of constraints in SAT can be very inefficient (see e.g. [100, 63]), and to handle same-directed cycles in the infrastructure, we add instead a refinement step around the SAT solver which searches each state for this kind of circular reasoning and adds a single constraint each time this situation appears.

The loop check procedure checks for each train t_i and for each state s_j, whether the set of routes R_i^j allocated to the train has any strongly connected components $\mathrm{scc}_i^j \subseteq R_i^j$ with $|\mathrm{scc}_i^j| > 1$, and in that case adds a new constraint to the SAT problem:

$$\bigvee \left\{ \neg(o_r^j = t_i^j) \mid r \in \mathrm{scc}_i^j \right\} .$$

Fixing these consistency errors gives valid plans even in the presence of same-directed infrastructure cycles, but even planning on infrastructure without cycles may cause repetition to appear in the dispatch plans. For example, at the end of a railway corridor, trains must be able to switch directions and go back to where they came from. In the description of dispatch planning above, if trains are allowed to stop and reverse their direction, the directed graph of routes becomes cyclic, and there is in principle an infinite number of different possible dispatch plans for any train movement.

Allowing trains to turn, and allowing loops in the infrastructure, will lead to the bounded model checking planning method finding more and more solutions when increasing the number of steps. Most of these solutions will exhibit some amount of repetition in the movement of trains, and this makes them of little value to the railway engineer. We suggest some different solutions to this challenge below, roughly ordered by how complex the implementation would be and how much quality would be improved:

- **Unlimited**: it could be feasible to have no limit on turning of trains, and no limit on the use of loops in the infrastructure. Since the bounded model checking technique will find the shortest plans first, they will often be the most valuable plans for the engineer, and the planner can be aborted when plans get too long and

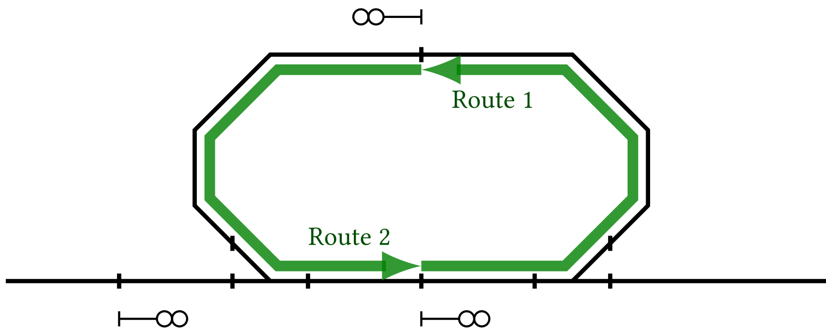


Figure 3.10: Example of cyclic infrastructure. Here, to ensure train path consistency (C2), additional constraints are needed over the acyclic case. Route 1 and route 2 both provide each other's justification for a train appearing there, possibly making an error of circular reasoning.

repetitive and as such are no longer valuable for the verification of the design. However, the fully automated verification tool would have to set a carefully considered upper bound on the number of plan steps.

- **Specified turning**: the specifications of the operational scenarios can be extended to include turning explicitly at visits. This increases the specification burden on the engineer, but ensures that there cannot be an unbounded number of distinct plans. However, it could also cause some plans to stay undetected if they require turning and the engineer did not think of it. Also, this method does not help the situation with loops in the infrastructure.

- **Bounded number of turns**: instead of writing out each turn explicitly, the capacity specifications could be extended to include an upper bound on the number of turns. The bound would have to be adjusted to balance running time and plan quality (low bound) with the possibility of detecting more complex plans (high bound).

- **State space repetition constraint**: to ensure that the whole state of the system does not repeat from one stage to another. This requires adding a constraint on each pair of states, which could make the SAT instance significantly larger.

$$\bigwedge_{0 \leq i < j < k} S_i \neq S_j.$$

  Such constraints may also be added lazily, i.e. by incrementally adding the constraints only when they are violated in a SAT solution (see [53]). This constraint would eliminate the possibility for an infinite number of distinct plans, but could still cause unnecessary repetition locally, since repetition in one part of the model could be accompanied by progress in another part of the model.

- **Repetition filtering**: even when the state as a whole does not repeat, there may be sequences of allocation to a subset of trains which can be considered repeating. We would like a more domain-specific definition of repetition, based on a graph analysis of the dispatch plans produced. This can be implemented by rejecting solutions which exhibit such repetition. We define this more carefully in the section below.

As we find the last option to be the most complete solution requiring no change to the specifications, we describe its implementation here in more detail.

3.4.3 Filtering out unnecessary repetitions

We now define the notion of *unnecessary repetitions* and show how to identify them on a given dispatch plan. First, we define the notions of *yield* and *repetition.*

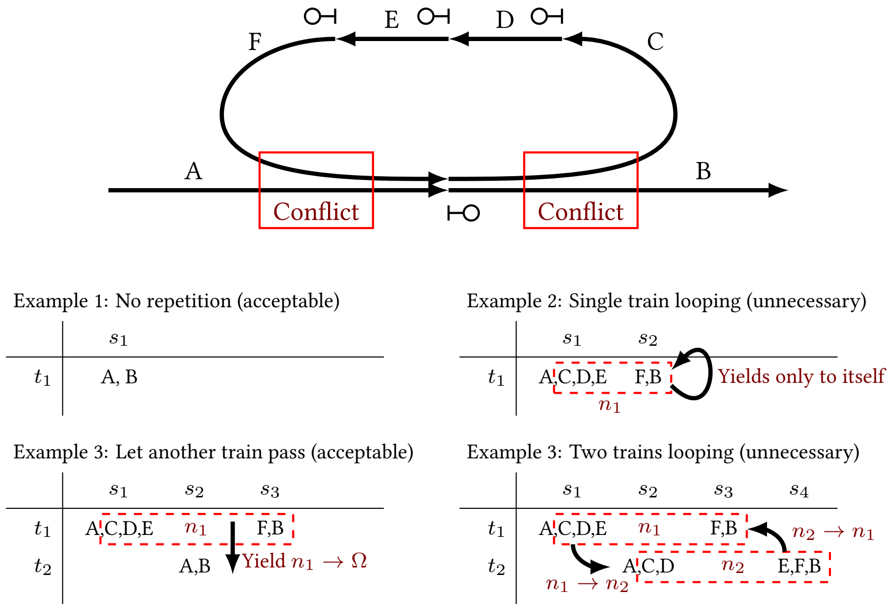


Figure 3.11: Examples of repetition justification using yields, demonstrating acceptable and unnecessary repetitions. Each of the routes A, B, C, D, E, and F shown in the infrastructure route graph is long enough to contain each train completely. Examples use trains t_1, t_2 and states s_1, s_2, s_3, s_4. Repetitions are shown as red dashed boxes, and yields are shown as arrows between repetitions.

A train t_1 *yields* to another train t_2 if t_2 is occupying a route whose resources are needed for t_1 to proceed (thereby allowing t_1 to defer its progress as defined in constraint (C8), Section 3.4.1.6). More precisely, if t_2 occupies some route r_2 in state s, and t_1 allocates a route r_1 in state $s+1$, where r_1 conflicts with r_2, we say that t_1 yielded to t_2 in state s.

Now, consider a train t that enters the model from some model boundary and exits through another boundary by traveling a sequence of routes $r_1, \ldots, r_{m+1}$, which we call the train's *path*. For each pair of consecutive routes r_i, r_{i+1}, the exit signal of r_i is the same as the entry signal for r_{i+1} (described as constraint C2 in Section 3.4.1.2), which we call the delimiting signal $u_i = \text{delim}(r_i, r_{i+1})$ between the routes r_i and r_{i+1}. We say that the train visits the sequence of signals $u_1, \ldots, u_m$ defined in this way.

A signal appearing several times in this sequence ($u_i = u_j$ with $i < j$) indicates a cycle in the train path. Let $s_a = \text{alloc_state}^t(r_i)$ be the state where route r_i

starting in u_i is allocated to t, and let $s_b = \text{alloc_state}^t(r_{j-1})$ be the state where route r_{j-1} ending in u_j is allocated to t. We say that the train t *repeats* on the interval s_a to s_b and write repeat(t, s_a, s_b).

In most cases, we would like to disallow such repetitions, but there are two exceptions. Firstly, if the train fulfils a specified visit on the state interval s_a to s_b (see constraint (C5), Section 3.4.1.5), the repetition is acceptable. Secondly, if the train yields to another train in a state s_y such that $s_a \leq s_y \leq s_b$, we say that the yield *justifies* the repetition. For example, if a train goes into a siding track to allow another train to pass by, the first train could reverse into the main track again to proceed, thereby performing a repetition that is acceptable. See Figure 3.11 for a few examples. However, if one repetition is justified by yielding to another train in a state which also has a repetition that is justified by yielding back to the first train, this does not make these repetitions acceptable. We would like to disallow such circular justifications, and we formalize this using the *yield justification graph*, $G = (V, E)$, defined as the directed graph where:

- The set of nodes N contains each repetition, repeat(t, s_a, s_b), and a special non-repetition node Ω.
- The set of edges E contains the edge $n_1 \rightarrow n_2$, where $n_1 = \text{repeat}(t_1, s_a, s_b)$ and $n_2 = \text{repeat}(t_2, s_c, s_d)$, whenever these nodes n_1, n_2 exist in N and t_1 yields to t_2 in a state s where $a \leq s \leq b$ and $c \leq s \leq d$.

  However, if $\text{repeat}(\mathrm{t_1}, \mathrm{s_a}, \mathrm{s_b})$ exists, and t_1 yields to t_2 in state $s_a \leq s \leq s_b$, but there are no matching repetitions n_2, then the edge $n_1 \rightarrow \Omega$ is included instead.

We say that a repetition is acceptable if Ω is reachable from the repetition's corresponding node in the yield justification graph. A repetition that is not acceptable by these two criteria, is an *unnecessary repetition*, and we discard the candidate dispatch plan and add a new constraint to the SAT problem to disallow it using the relevant component of the yield justification graph. This adds another kind of abstraction refinement to our algorithm, see Figure 3.12.

The methods for handling both loops and repetitions described here may cause performance problems on certain inputs. However, we have not encountered any real-world examples where this dominates the solver's performance.

3.5 Timing evaluation using simulation

For evaluating the behaviour of a railway system in full detail, there are various well-known simulation approaches which are routinely successfully used to analyse railway capacity. Because a simulation works by starting in a known state and applying known input to the system, it proceeds by executing imperative code to change the system state and to register event handlers to processes. Deterministic simulation models can handle very complex models in a short amount of time, but

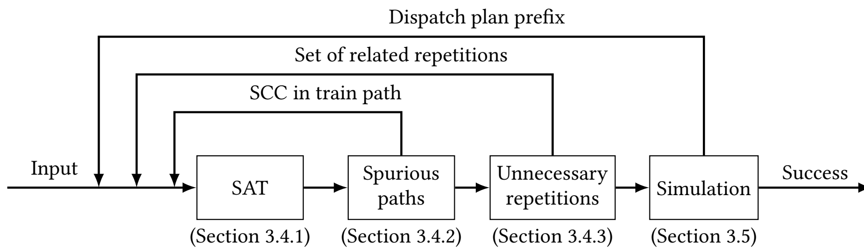


Figure 3.12: Main algorithm for local capacity verification (extended from Figure 3.4) with two more tests for handling loops and repetitions.

unlike a planning model, one cannot prescribe which state the simulation will end up in, only measure the outcome. Simulation methods are commonly used to develop and assess time tables, and by introducing stochastic elements in the model and repeating the simulation a large number of times, the robustness of a time table can be analysed (e.g., see [129]).

Discrete event simulation (DES) is a simulation technique based on assuming that changes to system state happen only at a set of discrete points in time, so that the simulation can progress efficiently by jumping from one point in time to the next point in time where an event is scheduled. This simulation assumption can be made to work even for the continuous dynamics of train movements, because we assume that each train's dynamics do not interact directly with other train's dynamics. Trains exist in separate worlds which are only connected to each other through the control system, and the control system has only discrete state changes. Each train acts separately on the information it has received from signals so far, and needs only to predict how long it will take to reach the next signal or sensor where it interacts with the control system.

In our tool architecture, the planner component works on an abstraction of the simulation problem that is just detailed enough to ensure that trains end up where they are specified to go, and that the system does not enter a dead-lock state. This is the reason that the planning model must include safety zones, partial release and the lengths of routes and trains – the sequences of routes and trains are represented exactly so that we know what to expect during the simulation. If it turns out that the planner's assumptions about where the trains end up does not work out correctly in the simulator, then the correspondence between planning and simulation is broken, which may be a modelling error in the simulator or errors in the route specifications, for example if the switches are configured to turn in the wrong direction. Running the capacity verification assumes that the route specifications are correct, and this may be verified through other means (see [168]).

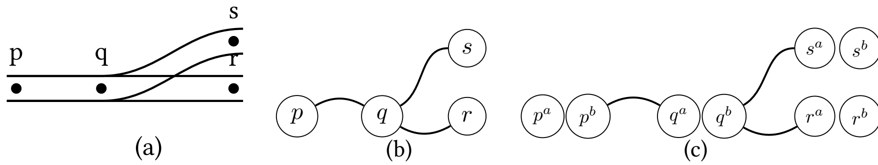


Figure 3.13: (a) On the railway network, paths p-q-r and p-q-s exist, in both directions. (b) In a conventional undirected graph representation, there would also be a path r-q-s. (c) When the graph is extended to include two sides of each node, there is no longer a path r-q-s.

3.5.1 Implementation

For our capacity verification tool for railway construction projects, we have implemented a simulation program using techniques described in [78]. We provide a brief overview here of the main components of this simulation program.

3.5.1.1 Infrastructure and interlocking specifications

For the purpose of developing a complete proof-of-concept capacity verification tool, we implemented a simplified simulation system for railways with main signals, detector, switches, routes, trains, partial release, safety zones and more.

The input of railway infrastructure consists of *nodes*, representing locations on the tracks where transfer of information between the infrastructure and the train can happen. Objects include switches, detectors, signal sighting locations, and points of discrete changes in track properties such as radius and gradient. Nodes are connected by *edges*, which have a specified length. Edges are not directed, because tracks can be traversed in both directions, so each train refers to the edge it is travelling on as an ordered tuple of nodes. For example, a train travelling from node a to node b will store its current location as either (a, b) or (b, a), depending on the direction of travel.

However, a simple undirected graph model also lacks the information that the train needs to figure out which edges can be followed while still travelling in the same direction. Representing this as a directed graph would require deciding on a *global* notion of direction, which is not directly compatible with cyclic infrastructure graphs where a train can travel forwards from one track and arrive at the same track, now in the opposite direction.

A more suitable data structure for simulating railway networks is the *double node graph* described in [116] where each node of a conventional graph is represented as a two linked nodes representing each of the two sides for approaching each track location, see Figure 3.13. A train reaching a node may only proceed by travelling on edges starting in the opposite node. Also, signals typically only apply

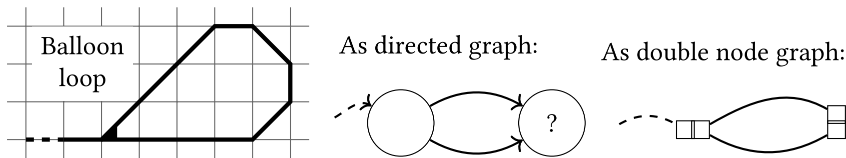


Figure 3.14: The balloon loop infrastructure is an example where directionality of travel cannot be suitably captured as a directed graph.

in one of the travelling directions, so a train passing a pair of nodes only reacts to the objects that are located on the *exit side* of the node pair. This model allows for a *local* notion of directedness, and avoids deciding on a global direction concept such as up/down or outgoing/incoming often used in railway engineering. A global directionality requires considering special cases to handle railway networks where a train's up/down direction may change without the train reversing its direction, such as the balloon loop example which is commonly seen on tram lines, see Figure 3.14.

3.5.1.2 Dispatching trains

From the planner we extract the following **dispatch plan**, which serves as the external events input to the simulation.

- **Start train**: start a new train process with given train parameters, initial velocity, and a route entering from a model boundary.
- **Activate route**: start a new route activation process for the given route.
- **Swing safety zone**: replace one active safety zone with another.

Note that the times at which these events happen are not given by the planner, only their order in time.

All the rest of the simulation output is determined from these inputs. The inputs start processes in the simulator, which may in turn start other processes. When all processes have finished, the simulation is done.

3.5.1.3 Dynamic infrastructure data

Our DES for railway simulation uses the following observable state:

- **Switches**: objects that fire events when they enter their left or right traversable state, and which can be called from the route process to start switching.
- **Detection sections**, objects that have an allocation and an occupancy status, which are observable through events.

- **Signals**: objects that have a movement authority length which can be observed by a train.

3.5.1.4 Processes

The core of the discrete event simulation technique can be implemented in a mainstream programming language as coroutine-like processes which can manipulate the state of the world at the current point in time or choose to wait for events which will be caused by other processes in the future. We used the Rust programming language, where coroutines are only experimentally available, so we used an explicit state machine model to represent the progress of each process. The overall simulation process maintains a global clock, and when all processes are waiting for events, the global clock is advanced until the next scheduled event.

The main processes in railway simulation are:

- **Elementary route activation** waits for resources, allocates them, sets switches to given positions and starts the following sub-processes:
  - **Release trigger**: listens to a *trigger* detection section which is designated as the release trigger for a partial route. After the detection section has first been occupied, and later freed, resources are released for use in other elementary routes.
  - **Signal catcher**: sets the route entry signal to the 'proceed' aspect, then waits for a given trigger section to become occupied before setting the signal to back 'stop'.
  - **Overlap timeout**: releases some resources after a given timeout. The timeout is started on the allocation of a specific track section (the *trigger*).
- **Swing safety zone**: replace one active safety zone with another. First, wait to allocate the additional required resources. Then release the resources which are no longer required.
- **Train** evaluates movement authority using information from signals currently in sight, and takes one of the following actions: accelerate, brake, or coast/wait. Braking curves from velocity limitations are calculated, representing the train driver's plan for when to start braking. We calculate a guaranteed minimum time until further action is required from the driver by taking the minimum time until one of the following happen (see also Figure 3.15):
  - train arrives at a new node
  - train reaches maximum velocity
  - train enters the area of a new velocity restriction
  - acceleration/coasting curve intersects the braking curve

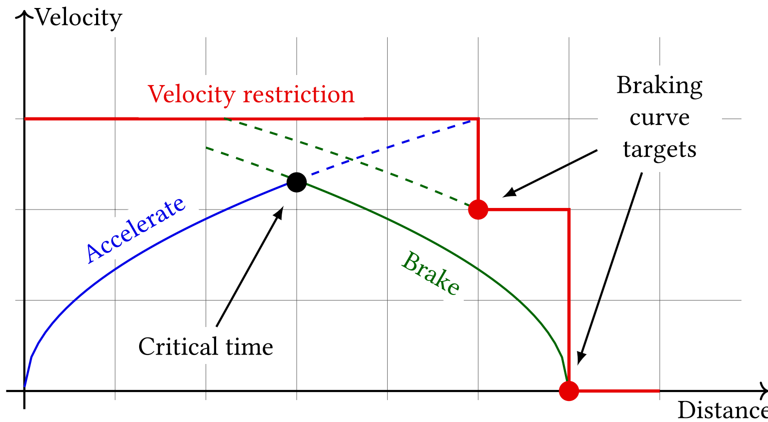


Figure 3.15: The train driver's decision about when to accelerate/brake/coast happens at intersections between acceleration curves, braking curves and velocity restriction curves. In this example, the train can accelerate until the critical time where the acceleration intersects with the braking curve towards the second velocity restriction ahead (the first one is not critical).

After this minimum time has passed, or any signals currently in sight have changed state, the train updates its position and velocity according to the chosen driver action and the laws of motion.

Note that since we assume a constant maximum acceleration and braking, the equations of motion can be solved analytically, and there is no need for discretizing the time or space domains, except for the re-evaluation of the equations of motion at discrete events. This ensures that the train starts braking in time using only the information available to the driver at any given time.

3.5.2 Extensions and alternative simulators

In our simulation model, trains re-calculate braking curves analytically on every possibly relevant event. This makes for a high-performance system, but in real-world engineering there are other complexities that we do not yet handle in this system, such as:

- More complex signalling and automated train protection systems.
- Local variations and details of infrastructure, such as the inner workings of components from different vendors performing various tasks like route allocation, de-allocation, safety zones, partial release, level crossings, etc.

- Train dynamics models using curve radius, gradient, air/tunnel resistance, weight distribution, etc.
- Stochastic variation in simulation output.

Our system can be extended with these features, or it would also be possible to swap out our simulation module with a more comprehensive solution or a commercially available offering (see [129, 104]), as long as this simulation program can be run in batch mode using the range of input described above. Also, implementing a discrete event simulation is most elegantly done through co-routines, such as in the SimPy[3] Python library, or though specialized languages for simulation such as ABS[4]. However, for the simple simulation system we have implemented, the number of distinct states in each type of process is so low that it can easily be managed by explicit state machine logic.

3.6 Case studies and performance

This section presents running times for different typical performance specifications on different types of railway infrastructure where the size and complexity of the model is typical for the scope of railway construction projects. Verification performance on various test examples as well as real stations is presented in Table 3.1. The table shows the time spent in each solver component, and also shows the number of invocations n_{DES} of the simulator, which is very low in most of the practical cases. This supports our hypothesis that the chosen abstraction and CEGAR loop is efficient. The two-track station used in Figure 3.1 is not too complex, having only 6 elementary routes. Even so, this scale is still interesting for verification in practice, since there are many possible mistakes to uncover.

The Norwegian railway infrastructure manager Bane NOR has supplied a infrastructure model in the railML format of the whole national railway network [138] from which we have extracted some more complex examples. Figure 3.16 shows cut-outs from the visual representation of these models, i.e., the stations Kolbotn, Eidsvoll, and Asker were converted from the railML models.

We have also tested against an infrastructure model from the Arna construction project that uses the RailCOMPLETE CAD design software, a realistic use case for agile verification.

Finally, to test the limitations of scalability in our method, we construct a set of examples where m stations each with n parallel tracks each are serially connected by a single track. In this case, when a timing bound is slightly too small to be satisfiable, the planner will have to come up with n^m plans for timing evaluation. This scenario is outside the intended use case for our method: path selection can

[3]See `https://en.wikipedia.org/wiki/SimPy`

[4]See `http://abs-models.org/`

Infrastructure	Property	Result	n_{DES}	t_{SAT}	t_{DES}	t_{total}
Simple (3 elem.)	Run.time	Sat.	1	0.00	0.00	0.00
	Crossing	Unsat.	0	0.00	0.00	0.00
Two track (14 elem.)	Run.time	Sat.	1	0.01	0.00	0.01
	Frequency	Sat.	1	0.01	0.00	0.01
	Overtaking 2	Sat.	1	0.00	0.00	0.01
	Overtaking 3	Unsat.	0	0.01	0.00	0.01
	Crossing 3	Unsat.	0	0.01	0.00	0.01
Kolbotn (BN) (56 elem.)	Run. time	Sat.	2	0.01	0.00	0.02
	Overtake 4	Sat.	1	0.05	0.00	0.06
	Overtake 3	Unsat.	0	0.05	0.00	0.06
Eidsvoll (BN) (64 elem.)	Run. time	Sat.	2	0.01	0.00	0.02
	Overtake 2	Sat.	1	0.08	0.00	0.08
	Crossing 3	Sat.	1	0.04	0.00	0.04
	Crossing 4	Unsat.	0	0.21	0.00	0.21
Asker (BN) (170 elem.)	Overtaking 2	Sat.	1	0.20	0.00	0.21
	Overtaking 3	Unsat.	1	0.73	0.00	0.74
	Crossing 4	Sat.	0	0.75	0.00	0.77
Arna (CAD) (258 elem.)	Run. time	Sat.	1	0.02	0.00	0.04
	Overtaking 2	Sat.	1	0.50	0.00	0.51
	Overtaking 3	Sat.	1	1.43	0.00	1.45
	Crossing 4	Sat.	1	1.73	0.00	1.74
Gen. 3x3 (74 elem.)	High time	Sat.	1	0.01	0.00	0.01
	Low time	Unsat.	27	0.18	0.01	0.19
Gen. 4x4 (196 elem.)	High time	Sat.	1	0.01	0.00	0.03
	Low time	Unsat.	256	2.08	0.26	2.34
Gen. 5x5 (437 elem.)	High time	Sat.	1	0.06	0.00	0.09
	Low time	Unsat.	3125	38.89	4.35	43.24

Table 3.1: Verification performance on test cases, including Bane NOR (BN) and RailCOMPLETE (CAD) infrastructure models. The number of elementary routes (*elem.*) is shown for each infrastructure to indicate the model's size. n_{DES} is the number simulator runs, t_{SAT} the time in seconds spent in SAT solver, t_{DES} the time in seconds spent in DES, and t_{total} the total calculation time in seconds.

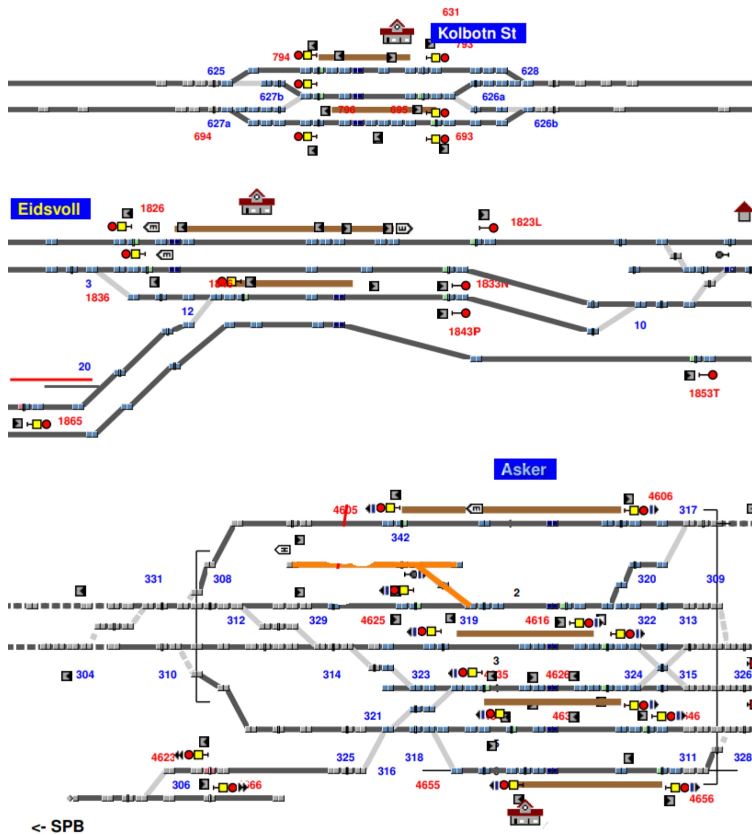


Figure 3.16: Stations Kolbotn, Eidsvoll, and Asker from Bane NOR's model of the Norwegian national network [138].

on this scale instead be based on static speed profiles. Capacity over many stations is better suited for the established timetabling tooling.

We attempted an alternative implementation using the PDDL+ solver SMT-Plan+, but found that even for greatly simplified models, the required number of steps and numerical constraints put all our case studies out of reach for sub-second verification times.

3.7 Related work

Railway timetabling and capacity analysis has often been posed as a planning problem and solved using mixed integer programming and similar approaches. Zwan-

eveld et al. [176] use integer programming on a problem closely related to our low-level railway infrastructure capacity verification problem. Isobe et al. [79] formulate a similar model in timed CSP, representing train locations, velocities, and control logic. Our definition of the problem above includes non-linear constraints on train dynamics (acceleration/braking power) and communication constraints (trains must slow down if they have not been informed of movement authority), which are relevant in construction projects but less relevant in timetabling.

Many variations on discrete event simulation are used in railway dynamic analysis, see e.g. [116, 78, 86].

In the planning literature, the PDDL+ language [59] has been introduced to capture mixed discrete/continuous planning problems such as the one studied in this chapter. General-purpose solvers have recently been developed, using time domain discretization (DiNo [135]) or the SMT theory of non-linear real arithmetic (SMTPlan+ [32]).

3.8 Conclusions and future work

Railway signalling have few design tools which allow rapid prototyping by anticipating the verification which is to be performed in later stages. Such tools are needed to improve quality and efficiency of the design process, according to Norwegian railway engineers of Railcomplete AS and Norconsult AS.

This chapter has demonstrated a control system design tool which can verify performance properties in the scope of a single project from high-level specification by synthesizing schedules. It automates the following activities:

- Detailed **running time** analysis – verify the time required for getting from point A to point B, taking into account train dynamic characteristics, communication constraints, and control system logic and latency.

- Detailed **schedulability** analysis – verify frequency of trains arriving at a station, and simultaneous opportunities for crossing, parking, loading, etc.

Our approach carves a new niche in the following sense: the level of detail supported by this tool is much greater than the traditional by-hand approaches for running time and schedulability analysis – and the amount of background data and work is much less than the whole-network stochastic operational analysis typically used for later-stage verification. To make the method approachable for engineers, the required input is the minimum of information required to verify the relevant properties. For example, the specific paths each train takes through the station is not an input, but different possibilities for realizing paths are explored by the verification procedure. This also makes the method more appropriate for early-stage design, where track lengths, topology, and component placement might be

adjusted to achieve design goals, and engineers can in this way get feedback on design choices without requiring large efforts to repeat the verification.

The goal of our suggested tool chain for railway engineering is (1) to allow fully automated performance verification and (2) use minimal input documentation for the verification. Both of these aspects encourage bringing in performance verification into frequently changing early-stage design projects, avoiding the costly and time-consuming backtracking required when later-stage analysis reveals unacceptable performance.

As future work we plan to integrate the current verification prototype tool in the RailCOMPLETE CAD environment and test the usability with the engineers using this tool in their design work.

4 Optimization and synthesis of signalling component layouts

This chapter presents an optimization-based synthesis method for laying out railway signalling components on a given track infrastructure to fulfil capacity specifications in the context of a construction project.

The main synthesis algorithm starts from an initial heuristic over-approximation of required signalling components and iterates towards better designs using two main optimization techniques: (1) global simultaneous planning of all operational scenarios using incremental SAT-based optimization to eliminate redundant signalling components, and (2) a derivative-free numerical optimization method using as cost function timing results given by a discrete event simulation engine.

Synthesizing all of the signalling layout might not always be appropriate in practice, and partial synthesis from an already valid design can be an alternative. In consequence, we focus also on the usefulness of the individual optimization steps: SAT-based planning is used to suggest removal of redundant signalling components, whereas numerical optimization of timing results is used to suggest moving signalling components around on the layout, or adding new components. Such changes are suggested to railway engineers using an interactive tool where they can investigate the consequences of applying the various optimizations.

4.1 Specification-based design synthesis

Building on the verification work presented in the two chapters above, we present here an optimization method where signalling components, i.e. mainly signals and detectors, but also balises, derailers, and catch points can be moved or removed from the design to improve capacity.

We show how our SAT-based planning procedure can be extended to find redundant signalling equipment, and how a simulator can be extended to move signalling equipment around using continuous-domain mathematical optimization methods and discrete event simulation. With the use of a heuristic initial design algorithm, the optimization procedures can be applied even if the user has not yet supplied any working signalling design, and in this way we get a synthesis algorithm. If a working design is already in place, our method suggests possible design improvements to the user of an interactive tool, so that the engineer has the final say in making changes to the design, and can investigate how the changes influence the infrastructure and operational scenarios.

These methods are a step towards a railway signalling engineering methodology based on explicit specifications, and using analysis and verification tools every step along the way, which we believe can improve decision-making.

The main goals of this chapter are: (1) suggesting and demonstrating a novel specification-based design methodology for the layout of railway signalling components, (2) extending existing planning and simulation methods to make changes in the designs which improve their quality with respect to given specifications, and (3) showing how incremental optimization and partial synthesis can be used in specification-based design through an interactive tool.

4.2 Design principles for local railway capacity

The basic safety principles used in most railways around the world are based on dividing railway lines into *fixed blocking sections*, and use signals and train detectors together in an automated interlocking system which prevents one train from entering a blocking section before it has been cleared by the previous train.

The block section principle directly impacts the maximum frequency of trains, and consequently the *capacity* of the railway, through the interplay between train parameters (length, acceleration and braking), track layout (how many tracks are available at which stations), and the location of signalling equipment. The topic of this chapter is how to choose the number and locations of signals and detectors to optimize capacity.

There are two main design methods for signal and detector locations, which have different application areas. The first method is the *blocking time diagram* where a single track on a railway line, or a single path through a railway station, is presented on the horizontal axis, and consecutive trains travelling the same path are plotted with the blocking time of each section shown as rectangles stretching out on the vertical time axis (see Figure 4.1).

The second design method is to use a schematic track plan showing the topology of tracks and the locations of signals, detectors, and other signalling system components. The schematic plan is not geographically accurate (for the sake of readability) but is annotated with travelling lengths between relevant locations, such as from one signal to the next signal or detector. This plan is used in the design of *route-based interlocking systems* to make assessments of the effective lengths of station tracks, safety distances from a signal to other tracks (so-called *overlaps*), and more (see Figure 4.2).

Note here how the the blocking time diagram and the schematic plan provide views in different dimensions: the blocking time diagram provides continuous time and a single spatial dimension but does not treat different choices of path, while the schematic track plan shows all paths at once, but does not directly show how a train would travel in time. The latter concerns *schedulability*, while the former concerns *timing*. For detailed signalling design, the decisions that impact the interaction

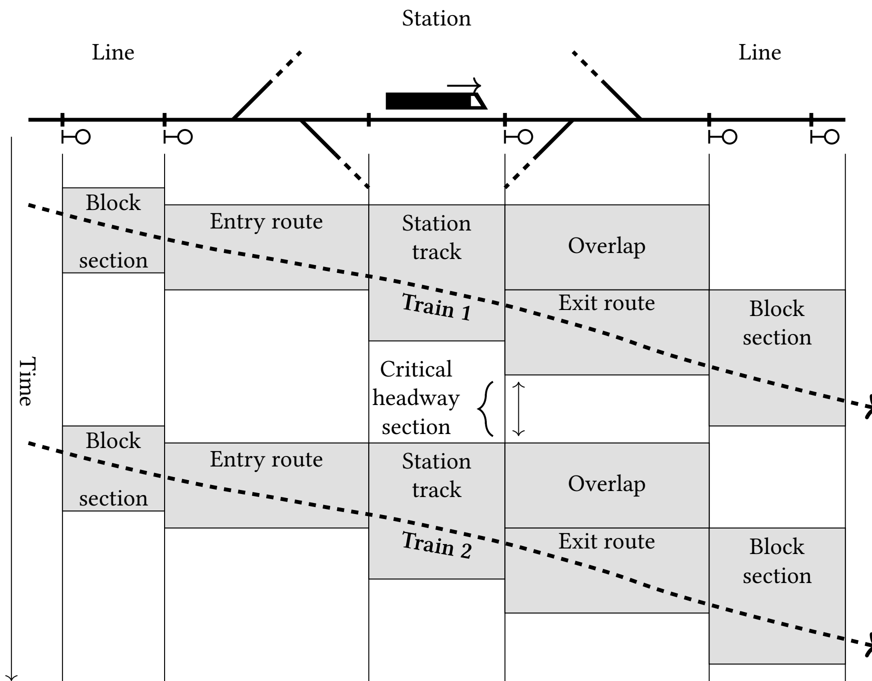


Figure 4.1: Blocking time diagram showing two (non-stopping) trains travelling from a line blocking section into a station and back onto a line blocking section. Dashed lines indicate train locations and velocity, and grey boxes indicate lengths and times of sections exclusively allocated to the trains. Figure adapted from [131].

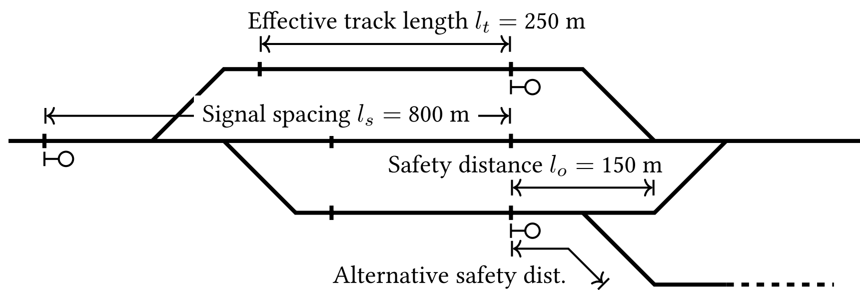


Figure 4.2: A schematic track plan, a key artifact in designing the signalling system in route-based interlockings. The plan is annotated with signalling components and distances between locations relevant for interlocking safety requirements.

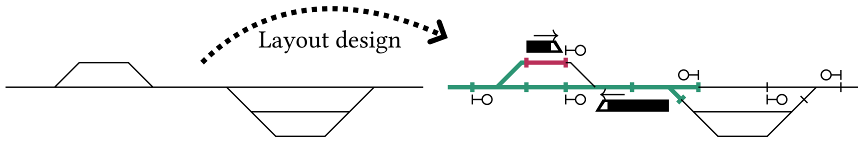


Figure 4.3: Railway signalling layout design places a set of signalling components on a given track layout to ensure that a set of capacity specifications can be fulfilled by dispatching trains in some way.

between these two analysis domains is a complex task where an engineer balances a high number of diverse concerns.

Placing signals, detectors, and other components so that trains can be guided to their intended tracks and platforms safely and efficiently is the problem of **railway signalling layout design**.

To be precise, we define the railway signalling layout design problem as follows: given a track plan, and a set of intended operational scenarios, decide on a set of signalling components (signals, detectors, etc.) and their locations, such that it is possible to implement a safe interlocking control system with which the specified operational scenarios can be dispatched efficiently.

We now consider the main constraints imposed on a signalling design, as first presented in the previous chapter, in the light of creating a design with capacity in mind:

1. **Physical infrastructure**: all the trains are guided by the rails and can only travel where the rails guide them. The space that trains move on is a graph with linear connections between nodes.

2. **Allocation of resources**: railway signals are connected to a control system called the interlocking, which ensures mutual exclusion of trains by reading from detectors and ensuring that signals can only signal movement authority when it is safe to do so. This entails that one can only allocate and free resources in certain groupings.

3. **Limited communication**: the most obvious way to improve capacity on an existing railway line is to install more signals to more finely subdivide the allocation of space so that trains can be travelling more closely on the line. However, since the train driver always has to be able to stop the train within the limits of the currently given length of movement authority, putting signals too close together will lower the speed that the train can travel with. This means that there is a limit to how many signals one can install before the capacity starts to decrease because of this (see Figure 4.5).

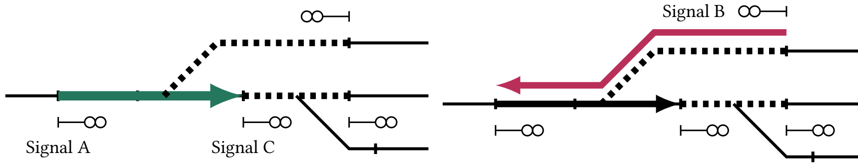


Figure 4.4: Allocation and freeing of resources can only be done within the limits of what information the control system can send and receive. In the left figure, a train travelling from Signal A must travel at least until signal C, and all resources in this path must be allocated and in a safe state before the train can proceed from A. Meanwhile, in the right figure, no train can proceed from Signal B because parts of the path require the same resources, which means that the elementary routes are conflicting and cannot be used simultaneously.

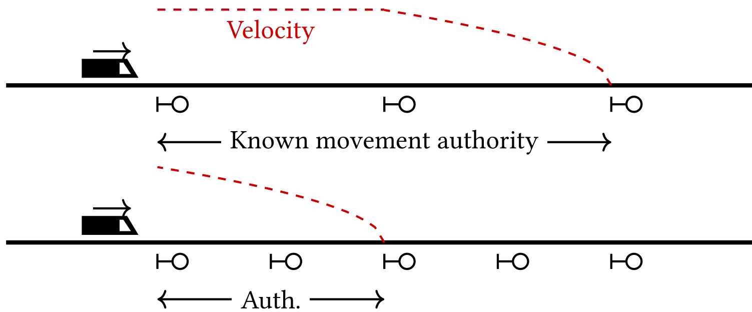


Figure 4.5: Signal information only carries across two signals (so-called *distant signals*).

4. **Laws of motion**: when a train is given a movement authority, this authority has a limited length and a limited maximum velocity. The driver must choose when to accelerate and brake to stay within the given authority.

$$v - v_0 \leq a\Delta t, \qquad v_i^2 - v_j^2 \leq 2bs.$$

In the methods for optimization and synthesis proposed below, we assume that the above constraints are absolute. In practice, engineers have subtle workarounds for each of these constraints whenever the situation requires a non-standard solution. Physical infrastructure (1) can often be modified by taking a step back in the planning process and re-evaluating the track layout together with track engineers. Allocation of resources (2) can be overcome by designing certain movements to be performed as shunting movements, i.e. a second-grade class of movement authority with lower safety requirements. Limited communication (3) can also be overcome by increasing the number of different aspects that the signals can com-

municate, or by using cab signalling giving additional communication between the interlocking system and the train driver. The ETCS Level 2 system currently being implemented in many European countries is capable of signalling any number of routes simultaneously through digital radio communication, effectively lifting the infrastructure-to-driver communication restriction. Finally, the laws of motion (4) cannot be overcome in themselves, but increasing the requirements for vehicles' acceleration and braking power may improve a layout design's expected performance.

4.3 Algorithm for synthesis and optimization

The following list is a summary of components in our work-flow for solving the railway signalling layout design problem:

1. **Track plan and capacity specification input**: track plans are graph-like structures with information about track lengths, boundary nodes, switches, and crossings, and we read this data from the railML format[1]. The capacity specifications are described in Section 3.3 above.

2. **Initial design**: a heuristic algorithm is used to over-approximate the signalling components required to plan any set of movements on the infrastructure, if they are possible with the given track plan. See Section 4.3.1 below.

3. **Derived interlocking specification**: we rely on an automatic derivation of interlocking specifications from the layout, allowing only customizations which are global parameters (e.g. overlap policy). Such derivation algorithms have been described in the literature, see [168].

4. **Planning optimization**: ignoring all timing aspects, we calculate the smallest set of signals and detectors that are able to dispatch all of the scenarios described in the local capacity specifications. This is done by solving a planning problem where all scenarios are planned simultaneously. An incremental SAT solver solves the plans and optimizes the number of signals that are used. See Section 4.3.2 below.

5. **Numerical optimization**: a measure for the performance of the design is calculated by dispatching all of the planned ways to realize the performance specifications and measuring the difference between the required time and the simulated time. This measure is used as a goal function for a meta-heuristic numerical optimization algorithm for moving the signals around, and when this algorithm converges, each track is tested for how much improvement would be had by adding signals to it and repeating the optimization process. See Section 4.3.3 below.

[1]See `https://railml.org/`

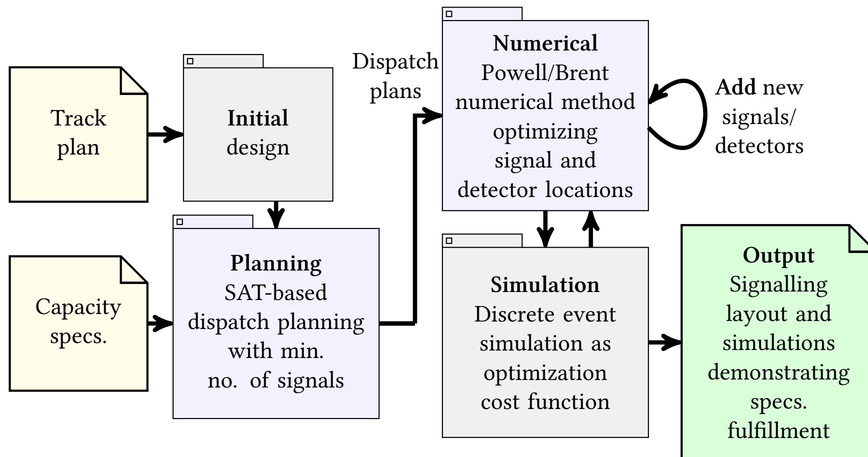


Figure 4.6: Synthesis process overview. Track plan and capacity specifications are given as input, and together with an initial design based on a heuristic algorithm they are given to the SAT-based planner for simultaneous dispatch planning of all usage scenarios. A numerical method takes the dispatch plans and adjusts the locations and number of signals and detectors until no better result from simulation is achieved.

6. **Output**: after the process is done, the user is left with a design and a set of dispatch plans and simulated train movements which describe how the capacity requirements are fulfilled by this design.

4.3.1 Initial design

When starting from an empty set of signalling components, most operational scenarios are not possible to even dispatch, because the railway interlocking safety principles require detectors and signals to have control over movements for safety purposes. Instead of searching for signalling components to add to the design to allow dispatching to happen, we start the synthesis procedure by heuristically over-approximating the components required to perform dispatch. We insert a signal and a detector in front of every trailing switch, and at a set of specified lengths corresponding to the choices of length of safety zone. We also insert a detector in front of every facing switch. See Figure 4.7. If more than one train is required on the same track for overtaking or crossing, we can also choose to insert signals at multiples of the trains' lengths. When there are several paths of the specified length leading to a trailing switch, we put signals and detectors at all the relevant

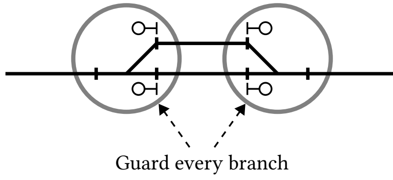


Figure 4.7: Initial design: put signal in place before every trailing switch, i.e. where tracks join together.

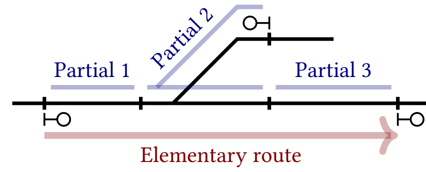


Figure 4.8: The planning abstraction of the train dispatch allocates a set of partial routes to each train. Elementary routes are sets of partial routes which must always be allocated together.

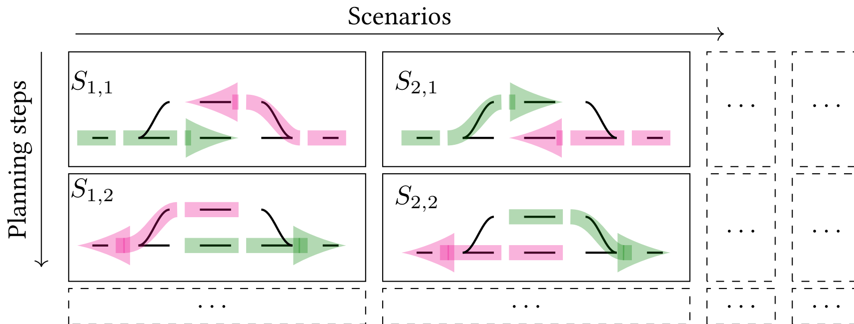


Figure 4.9: The planning matrix consists of the occupation status of a set of partial routes for each state required for dispatch planning, and for each scenario in the local capacity requirements. The top left cells show an example dispatch of a crossing movement where green areas show track segments which are currently occupied by a train going from left to right, while the pink areas show track segments which are currently occupied by a train going from right to left.

locations. This design aims to allow all possible dispatches and we rely on the next stage of the synthesis to remove redundant equipment.

4.3.2 Planning optimization

The operational scenarios of the local capacity specifications describe train movements only declaratively, so the first step to analysing concrete states of the system is to solve a planning problem which gives us a set of dispatch plans, i.e. determining sequences of trains and elementary routes which make the trains end up

visiting locations according to the movements specification.

Instead of using a constraint solver system (e.g. SMT solvers) to solve for route dispatching and train dynamics simultaneously, we have chosen to separate the *abstracted planning problem* (i.e. selecting elementary routes to dispatch) from the physical constraints of train dynamics. This choice was made for performance and extensibility reasons (see Section 3.4 above).

To find a subset of the signalling components from the initial design that is sufficient to successfully plan all the dispatches, we use the planning approach described above and add a set of signal usage Booleans u indicating whether the signal is needed. The set of occupancy status Booleans o is repeated once for each operational scenario, resulting in a SAT instance with parallel execution of each scenario on copies of the same infrastructure (see Figure 4.9). We link the signal usage status u to each copy of the state so that the signal is marked as needed if it is used independently of other signals:

$$\forall i \in \text{State} : \forall s \in \text{Signal} : \forall t \in \text{Train} : \quad \neg u_s \Rightarrow$$
$$\bigvee \left\{ \left(o_r^i \neq t \wedge o_r^{i+1} = t \right) \mid \text{exit}(r) = s \right\} \Rightarrow$$
$$\bigvee \left\{ \left(o_r^i \neq t \wedge o_r^{i+1} = t \right) \mid \text{entry}(r) = s \right\}$$

Similar approaches are taken for other signalling component types.

Now we find the smallest set of signalling equipment which is sufficient to allow dispatching all scenarios. We use a simple technique to minimize number of signals: take the sum of u variables as a unary-encoded number (see [15]) and solve SAT incrementally with a binary search on the upper bound on the sum.

4.3.3 Numerical optimization

When we have a design where dispatching is possible, we have fulfilled the discrete part of the dispatch plan. However, timing constrains might not yet be fulfilled, and we might also want to improve on the total execution time of the various dispatch plans. To improve on the basic design found by the planner, we solve a numerical optimization problem with a cost function f defined as a weighted sum of dispatch timing measures:

$$f_b(\overrightarrow{x}) = \sum_s w_s \left(\frac{1}{n_s} \sum_d t_{b+\overrightarrow{x}}(d) \right),$$

where $\overrightarrow{x}$ is a vector with components representing the location of each signal and detector, s indexes operational scenarios from the set of capacity specifications, w_s is weight assigned to the operational scenario, d indexes the set of n_s alternative dispatch plans derived by the planning algorithm for each operational scenario,

and $t_{b+\vec{x}}(d)$ is the time measure calculated by executing the dispatch plan d by discrete event simulation on the infrastructure constructed by adding the signal and detector locations $\vec{x}$ to the base track plan infrastructure b.

We define two basic operations for optimizing the timing performance of a signalling layout:

1. Searching for the optimal signalling component locations $\vec{x}$ for a fixed set of components located on a fixed set of tracks in a fixed order using Powell's method and Brent's method of derivative-free numerical optimization.

2. Adding a new signal or detector to any track.

4.3.3.1 Powell's method and Brent's method

Since we use simulation to measure the cost of a design, we do not have an expression for the derivative of the cost function f_b, and this function is not even guaranteed to be continuous. Even so, it is possible to use numerical methods for local optimization without taking derivatives. We used Brent's method for minimization in the single-parameter case, with the generalization to multivariate functions by Powell's method.

Powell's method works as follows: given a domain $D \subset \mathbb{R}^n$, an initial point $\vec{x}_0 \in D$, and a cost function $f : D \to \mathbb{R}$, create a set of search vectors V initially containing each of the unit vectors aligned with each axis of $\mathbb{R}^n$. Iterate through the search vectors $\vec{v}_i \in V$ and do a line search for the parameter α giving the optimal point of $\vec{x}_{i+1} = f(\vec{x}_i + \alpha\vec{v}_i)$. After updating $\vec{x}$ using each search vector, remove the search vector which yielded the highest α and add instead the unit vector in the direction of $\vec{x} - \vec{x}_0$. See [24] for details.

Brent's method for optimization is used for the line search sub-routine in Powell's method. It takes a range of α values for which $\vec{x}_i + \alpha\vec{v}_i$ is inside D, and does a robust line search which finds a local minimum even for non-smooth and discontinuous functions. The method keeps a set of the three best points seen so far and fits a quadratic polynomial with the three best function values as parameters (called *inverse quadratic interpolation*). If the predicted optimum by the quadratic fit falls within an expected range, it used as the new best guess, otherwise the method falls back to golden-section search. See [125, 24] for details.

To simplify the use of the numerical algorithms, we map each signalling component's position to an intrinsic coordinate in the interval $[0, 1]$, so that the vector $\vec{x}$ keeps within $D = [0, 1]^n$. For a component with position p relative to the start of its track, if the component is the only component on a track, we define its intrinsic coordinate as

$$x = \frac{p - (l_a + l_{\min})}{(l_b - l_{\min}) - (l_a + l_{\min})},$$

where $l_a = 0$, l_b is the length of the track, and $l_{\min}$ is the minimum spacing between components. When there are several components on the same track, we convert the coordinates by processing the components in order of increasing p, and adjusting l_a to correspond to the location of the previous component on the track. In this way the whole of $[0, 1]^n$ represents valid component positions and we do not have to apply constraints to the search space by other methods.

See Figure 4.10 for an example of signalling components being moved.

4.3.3.2 Adding new components

When the above optimization has converged for a fixed set of components $\vec{x}$, we iterate over each track (and each direction), adding a new component and including its dimensions in $\vec{x}$, re-running optimization, and see which track, if any, most benefits from adding a signal or detector.

4.3.3.3 Discrete event simulation

The time measure t is calculated by simulation on a fixed infrastructure, which is a well-established method in railway capacity research. We have developed a simple custom simulator which we will not describe in more detail here (see [147] for a methodological overview, and [78, 25, 87] for discrete events simulation for railway applications). Commercial railway simulation software can also be used instead of custom solutions.

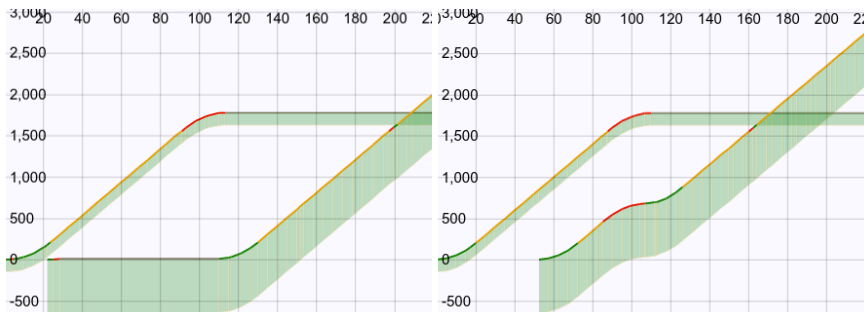


Figure 4.10: Partial screen capture from our interactive design tool showing before (left) and after (right) improving signal and detector locations for a two-track station on an overtaking scenario. Note that the time axis is horizontal in this example. A signal at $x \approx 0$ m is moved to $x \approx 700$ m so that the overtaking train is unblocked at an earlier time, lowering the overall time taken to perform the operation.

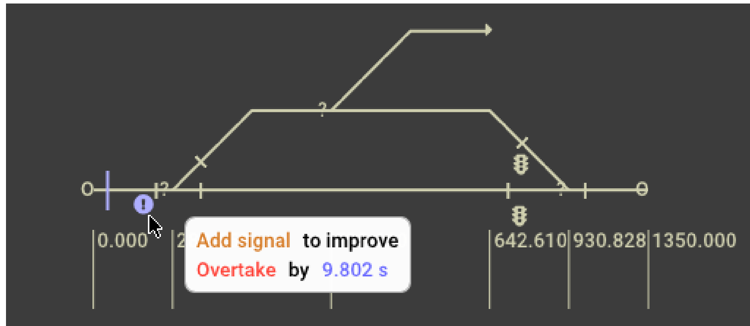


Figure 4.11: Partial screen capture from our interactive design tool showing suggestions for design improvement to the user, inspired by integrated development environments used for programming. The individual optimization steps run their calculations as a background process, showing an information symbol where the algorithm is able to provide an improvement over the current design. The user can decide to implement it or to dismiss this change and similar changes from future suggestions.

We also use an automated derivation procedure for interlocking specifications to adjust the behaviour of the control system after making changes in the infrastructure, similar to the procedure described in [168].

4.4 Local optimizations and interactive improvement

There are many reasons that a from-scratch synthesis can be unsuitable in practice. The main reason would be that the synthesis method itself is inadequate, for example if it fails to recognize a key concern that the design should be based upon, or if its calculation time prohibits practical use.

Even if the specifications successfully capture the capacity requirements, and the the synthesis algorithm in itself can adequately come up with designs with good capacity, there in practice often other constraints which can make a full from-scratch synthesis unsuitable. For example, in upgrade construction projects, it might be more useful to search for and suggest small changes which would be the most effective remedies for bottlenecks in a station's capacity.

To increase the number of ways that our methods can be useful, we consider also each optimization step as described below as a possible incremental step towards a better design, which can be performed by a user interactively. Using a computer-assisted design program for railway, or a drafting program (such as AutoCAD) extended with semantic information about railway objects and rail network topology, the user gets suggestions for smaller changes to their design and

can investigate how applying these changes affects the various scenarios.

Local optimization steps suggested to the user are the following:

- **Redundant equipment**: if removing a single object from the drawing can still be made to satisfy all local capacity requirements, the program suggests that the object is redundant. This class of suggestions is based on the SAT-based component minimization technique described above.
- **Local move of equipment**: if moving a single object or a set of nearby objects can improve the overall capacity measure on the station, the program suggests moving the object (or set of objects). This class of suggestions is based on the numerical timing optimization technique described above.
- **Adding equipment**: if adding a single piece of equipment (and performing local moves of equipment afterwards) can improve timing, the program suggests this to the user. This class of suggestions is based on the numerical timing optimization technique described above.

When the user accepts any of these changes, they can investigate how the dispatch plans and the timings change. The tool meanwhile calculates new suggestions based on the new layout.

We have developed a prototype tool which can calculate and suggest such changes to a user while they are editing their layout, and we are currently starting testing of this tool in an industrial setting together with railway engineers to investigate how useful such suggestions are, and how often they can be used compared to a from-scratch synthesis.

4.5 Related work

Although the literature is comprehensive on railway engineering in general, the safety-critical implementation railway interlockings, and operational analysis of large-scale railway networks, the signalling layout problem in itself has little coverage. We are only aware of the following works: Mao et al. [111] presented a genetic algorithm solution to signal placement, but the method is limited to the one-dimensional railway line, and does not handle signal placements inside stations/interlockings. Dillmann and Hähnle [47] describe a heuristic algorithm for upgrading German conventional signalling systems to an ETCS system, aiming to replicate the behaviour and capacity of the existing system.

4.6 Conclusions and future work

We have presented a method for partially or fully automating signalling layout design using SAT-based planning and discrete event simulation. The automation

of verification, optimization and synthesis rely on specifications that are tailored to the relevant scope, and we hope that this is a step on the way to integrating explicit formal specifications into the layout design process.

Our planning algorithm uses fixed blocks, so it handles conventional lamp signalling and the European standard ERTMS/ETCS Level 2, while handling Level 3 (which uses moving block) would require changes to the planning algorithm.

The simulation paradigm is imperative, progressing by calculating train trajectories forward in time, which makes the overall synthesis easily extensible with timing-related details, such as engine and braking power models, resistance models, operational regulations, automatic train control systems, etc. which do not impact the applicability of the dispatch plan but impact the timing performance.

Although our method is capable of making good design choices in several simple models, we are aware of several limitations. Firstly, the method is not complete – we cannot guarantee finding an optimum because of the following: (1) the initial design does not guarantee maximum possible schedulability, (2) although the global simultaneous planning is exact in finding the smallest subset of the initial plan which can dispatch the operational scenarios, this set might not be the optimal starting point for timing optimization, and (3) the cost that we use for numerical optimization can have multiple local optima, especially when summing the score for competing operational scenarios, in which case the method described above is not guaranteed to find the global optimum.

We have also identified the following concerns for scalability of the method: (1) the specification language is practical to use for passing tracks, junctions, and medium-sized terminal stations, but on large-sized terminals and larger-scale analysis across multiple stations, the language is not easy to use because it specifies single movements separately, (2) optimizing the number of detectors in the SAT problem requires quantifying over all paths, which will cause scaling problems on larger track plans with many path choices, and (3) the algorithm for adding new signals to improve performance is naive, and will be expensive for track plans with a large number of tracks.

Controlled natural language specifications

5

Automated formal verification techniques have the potential to greatly increase the efficiency of engineering. However, verification engines are not easy to take up in industrial practice. Even if the verification process is fully automated, integrating the tools into the users' workflow and formalizing properties and models requires careful thinking and domain expertise. The gap between automated verification and domain expert users is often caused by the lack of user involvement. The users are usually not experts in verification techniques, i.e. they do not know how to write properties in the verifier's language, nor how to build models for the verifier, nor how to interpret the output of the verifier when violated properties are found. In our case, the users are expert engineers from the railway domain, designing railway infrastructure.

We want to allow the end users to participate in the whole verification process. Firstly, the domain experts need to understand the verification properties that the tool actually verifies, as well as the model of the system that the tool works with. Secondly, we want to allow the users to actively participate in maintaining the verification properties, i.e. to change and adjust them when needed.[1] Thirdly, we want that the domain experts are able to create their own specifications and feed these into the verification engine, e.g. define specific expert knowledge as verification conditions.[2]

Involving the user in the design of a system is well-studied in the field of participatory design [154, 89]. We use the term *participatory verification* when talking about methods for including the end user in the verification process. The goal is to make automated verification techniques accessible to engineers with little programming or verification experience.

The efficient verification and troubleshooting tools presented in Chapter 2 performs a lightweight type of verification which we call *static infrastructure verification*, and the results are updated continuously as the engineer is modifying the station (see Figure 7.4 and Section 7.2). However, the Prolog-like formal logical specification language that we used for describing railway rules and regulations is not easy for inexperienced programmers to write. Ideally, railway engineers should be able to read the logical specifications to ensure that they correctly represent the engineering domain. Furthermore, engineers should themselves be able

[1] Authorities typically make small adjustments to regulations several times per year, whereas engineering best practices can be revised at any time.

[2] Such expert knowledge is often seen as proprietary valuable assets of the company.

to maintain and extend the rule base with limited support from verification experts. When we evaluated our verification prototype with railway engineers from RailCOMPLETE AS[3], they raised yet another concern: how could they trace the violation, which the tool displays graphically, back to the regulations documents?

These observations have led us to develop a controlled natural language (CNL), which we call RailCNL, meant to be used as an intermediate representation between natural language texts (i.e. the railway regulations) and Datalog [165] logic programs. RailCNL aims to be human-friendly enough for our domain experts to work with to overcome the above challenges, and thus getting them involved in using and improving the automated verification tool. At the same time, the language is a formal language which can be automatically translated into Datalog.

The rest of this chapter is organized as follows:

- We propose participatory verification as an agenda for making verification methods more accepted by their intended users. Section 5.1 describes a general view on participatory verification, and the rest of the chapter is concerned with applying this this view in a specific use case, namely the writing of verification properties by engineers, with the proposed solution of using controlled natural languages.

- We present a methodology for designing controlled natural languages for verification, described in Section 5.2, and the RailCNL made specific for railway regulations and specifications, which is described in detail in Section 5.3 and evaluated in Section 5.4.

- To fully evaluate the usefulness of using controlled natural languages for verification properties, we present two tools making use of RailCNL in Section 5.5. The concluding Section 5.6 contains some more related works and suggestions for possible continuations of this work.

5.1 Participatory verification

We propose to adopt techniques from participatory design [154] into formal verification processes. We try to convey here how the formal verification process can be seen as a participatory design process, pointing out what stages and components from verification can be enhanced by participatory design ideas, so to make verification more user friendly, and to properly include the user in those verification tasks where their participation is needed.

Formal verification techniques are developed by theoretical computer science researchers based on mathematical models, like automata, and logical formalisms, like temporal logics, aiming to verify complex properties of complex systems, where

[3] http://railcomplete.no

human reasoning or testing techniques cannot encompass the large amount of details. The formal verification technique is implemented into a verification engine which takes as input:

- a *model* (i.e., the system with some details omitted) and
- a *property* to be verified;

and returns an answer:

- either *correct* when the model satisfies the property, or
- *error* when the model violates the property and
- an *explanation* of why the violation happens.

A plethora of verification engines exists, each for different kinds of systems (avionics, railways, software, microchip) and different kinds of properties of interest (deadlocks, safety, availability, correctness). Many of these have reached enough maturity to be usable in industry on more than proof of concept scenarios. A main impediment in the wider adoption of verification engines and techniques is the high level of specialized expertise that is required in order to:

- build models that the engine will accept and work nicely with,
- write properties,
- understand the output explanations of these engines,
- understand what the verification engine does (how the verification algorithm works) so to increase trust.

Usually those that can perform these tasks are the same experts that also built the respective verification engine, with knowledge of specific types of logics, specific kinds of mathematical objects representing the models, and deep understanding of the verification algorithms employed in order to be able to decipher the outputs of the verification engine and know how to use them to repair the system problem discovered by the verification process.

However, one would want the users of the verification engine to be the respective engineers that design and work with the respective real life complex systems like avionics designers, software engineers, railway engineers. These are persons with high level of skill in their own field, but not in the field of verification. We want persons that work with defining the specifications for the complex system and the various safety regulations and standards, to be the ones writing also the verification properties. We want those that build the complex systems to be able to run the verification algorithm, understanding what it is supposed to do for them,

on the models provided by their peer engineers on the properties provided by regulators, and to understand the explanations of the verification tool when properties are not satisfied, i.e., to be able to use these explanations to debug their implementations and designs.

Like in any other product design process, the verification experts should go out of the loop as soon as they have finished designing and developing a specific verification engine (e.g., a verification tool built for some specific avionics system and specific kinds of safety properties as commissioned by some specific company). The verification engine is the product, and the users are the various field engineers. The users should be able to use a product in their daily tasks without any help from experts. However, since the final product is such a complex system, we need additional tools around the verification engine that would help the users participate in the verification process without needing interaction with verification experts.

Therefore, in participatory verification we define two phases and two main classes of software components.

Phase D: The design and development of the verification method and engine for the specific verification task (or area).

Phase V: The verification time when field engineers use the verification tools to debug and check their designs against specific regulations/properties.

Verification software components are seen as organized into:

Expert components include the verification algorithm and engine, various optimization modules, various translation tools between various specific mathematical structures as input to these modules and engine, including models like automata and property description languages like temporal logics.

User components include the UI tools, e.g., for displaying models, for writing properties, various modules for interacting with existing field tools and their UIs, various graphical/diagrammatic languages, editors for domain specific languages.

We observe that usually most of the resources are spent on expert components, whereas the user components are often disregarded in favour of hiring verification experts to use the expert verification components in doing the verification tasks during the verification phase. If some verification tool suite is becoming so popular as to attract companies, then investments in user components appear. The rest of this chapter is *concerned with Phase V and User components.*

In participatory verification the concept of *"participation"* comes in two flavors:

(i) Users participate during the Phase D in usability studies [50], helping the verification experts to develop a tool best fit for the specific verification task and for the field in which the engineers are supposed to use the new tool.

(ii) Users actively participate in the verification process, during Phase V, to define the models and properties for the verification engine and to interact and understand the outputs from the engine.

Traditionally, participatory design (or interaction design) is concerned with Phase D of building a product until its delivery on the market, and less concerned with Phase V, when the product is used, unless subsequent versions of improved products are planned.

It is a particular challenge of participatory verification to achieve the second form of participation because of the complexity of the product. It is often the case with verification tools that the difficult learning curve required to use them is too big for the field engineers to overcome. Because of this, too many good verification algorithms and methods are not adopted.

Participatory verification aims to increase adoption of verification techniques, making two fundamental observations:

Sympathy for the verification tool: if the end user is involved in the development (specification, testing, etc) of a complex tool for them (thus prone to seeing the bugs along the way), then the user will be more aware of which features of the tool are difficult to implement, and thus buggy, and which work well.

Empathy for the intended user: if the developer works with the stated intention of making the tool *for the end user*, knowing the capabilities of the end user, how she normally will use the tool, spending enough time on tailoring the tool to the actual expressed user needs, then the user will require little learning and effort for using the new tool with her normal working methods, also making use of all the features of the tool.

In the work presented in the chapters above, we have had the role of the developers, building a verification engine for railway designs. We have tried to follow the *Empathy* guidelines, working closer with the engineers and integrating our engine into the engineers' design tools. We observed their working procedures and tools as well as interviewed representatives. However, we did not achieve the *Sympathy* goals, one of the major impediments being the opacity of the verification method, including the encoding of the regulations that our engine was working with. Moreover, it was clear in the end that the engineers would not be able to write or change any of the verification properties by themselves.

The remainder of the section presents our solution for allowing the user of a verification method to participate in the definition of the properties to be checked. The rest of this chapter goes into further details, presenting first a methodology that we devised and followed for building such front-ends, then defining an actual constrained natural language that we called RailCNL, and finishing with the way we use RailCNL in practice both for reading and writing properties.

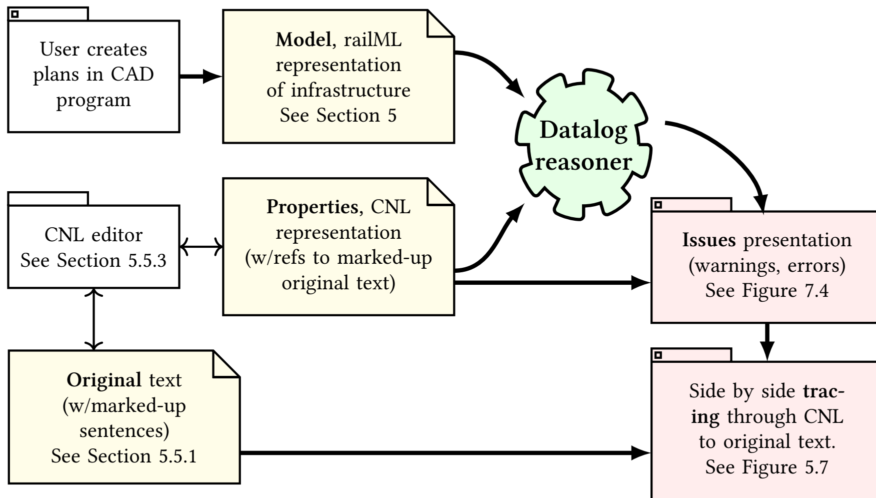


Figure 5.1: Verification process overview. *Models* come directly from the CAD program, which engineers are already familiar with. *Properties* come from paraphrasing the regulations using CNL, which in turn are translated into Datalog. The reasoner outputs *issues* (warnings and errors) which are presented to the user in the CAD program by highlighting the objects involved in the violation. Issues are traced back to the original text (i.e. the regulations) though identifiers on the marked-up sentences.

5.1.1 Participatory verification for railway regulations

To promote participatory verification of infrastructure railway designs against regulations, we design a property specification language for expressing railway regulations and expert knowledge, integrating it with our previously developed verification engine. Figure 5.1 presents the overall workflow of using the property language with special-made tooling, integrated with the engineer's CAD-based environment and our verification engine. Specifically, railway infrastructure static verification requires:

1. *Models:* railway infrastructure plans, typically created by arranging the station layout using CAD-based programs, e.g. extensions of Autodesk AutoCAD.

2. *Properties:* regulations and expert knowledge, extracted from regulatory and best-practices documents.

The formalization of these into Datalog, described in Chapter 2, which allows efficient automatic reasoning. The reasoning happens continuously, in the background, while the user is working on the CAD drawing. Violations of regulations and best practices are presented to the user in the CAD program graphically, on the drawing.

We are not concerned with the model because in our case it is automatically generated from CAD drawings, which is already the tool of choice for engineers, thus they are actively involved in making the models while drawing in the CAD-based RailCOMPLETE framework.

Describing verification properties using logical rules in Datalog is not new (along with other logics like temporal [9] or dynamic logics [71, 11]), and we expected that the declarative style of Datalog would make it easy for railway engineers to read and write such properties. However, a pilot project with the RailCOMPLETE engineers showed that they were not proficient enough in logic programming to understand our encodings.

To allow the engineers to participate in the verification process, we develop the controlled natural language RailCNL for representing properties on a higher level of abstraction, making them closer to the original text while still retaining the possibility for automatic translation into Datalog. This approach has the following advantages:

- RailCNL is domain-specific, i.e., tailored both to the types of logical statements needed by the verification engine, and to the regulations terminology. This allows concise and readable expressions, increasing naturalness and maintainability.
- The language closely resembles natural language, and can be read by engineers with the required domain knowledge without learning a programming language.
- A separate textual explanation (such as comments used in programming) is not needed for presenting violations textually, as the properties are now directly readable as natural text. Comments could still be used, e.g., to clarify edge cases or to clarify semantics, as is done in the original regulations texts where commenting is needed since the expected natural semantics of some regulations needs confirmation in certain cases (e.g., “yes, this rule applies even when (...)”).
- Statements in RailCNL can be linked to statements in the original text, so that reading them side by side reveals to domain experts whether the CNL paraphrasing of the natural text is valid. If not, they can edit the CNL text.

5.2 Design methodology for a verification front-end language

A controlled natural languages (CNL) is a constructed language resembling a natural language (such as English) but with added restrictions on its grammar and

vocabulary. The restrictions are typically aimed at reducing the ambiguity and complexity of unrestricted natural language. A CNL may or may not also be a formal language, depending on its intended use. Wyner et al. [175] give high-level recommendations on how to design controlled natural languages ranging from informal to formal, general to domain-specific, simple to complex. For a recent survey of CNLs, see Kuhn [94], whereas in Section 5.5.2 we survey CNL editors and their properties.

5.2.1 Using the Grammatical Framework to build CNLs

Grammatical Framework (GF) is a programming language for multilingual grammar applications [140]. A GF program defines a grammar consisting of an *abstract syntax* and one or more *concrete syntaxes*. The project also features the *resource grammar library* (RGL), which is a comprehensive linguistic model of natural languages with a unified API for forming sentences, and implementations of this API for 32 languages. The RGL encapsulates the linguistic complexity of the underlying natural languages, minimizing the effort needed to map an abstract syntax into another natural language, often reducing to simply providing the domain-specific vocabulary. This makes GF a valuable tool for building CNLs.

An abstract syntax consists of categories and constructors (functions), corresponding to a set of algebraic data types, which define the abstract syntax tree (AST) of the language. The following is an example of abstract syntax used to form sentences about distance restrictions on railway objects:

```
abstract Railway = {
  cat Object; Length; Restriction; Statement;
  fun
    Signal, Switch, Detector : Object;
    LengthMeters : Int -> Length;
    GreaterThan, LessThan : Restriction;
    ObjectSpacing : Object -> Object -> Restriction
      -> Length -> Statement; }
```

To express that signals should not be closer than 20m from a switch, we write:

AST:
```
ObjectSpacing Signal Switch
  GreaterThan (LengthMeters 20)
```

The concrete syntax creates a mapping from the tree-structured abstract syntax to text. Applying this mapping is called *linearization*. GF concrete syntaxes are invertible so that the concrete syntax also defines a *parser* for the language. This inversion is complete except for situations with ambiguities in the concrete syntax. Therefore, and especially when designing formal language front-ends, it is essential to limit the possible ambiguities in the language to get an exact correspondence

between the concrete language (linearization) and the AST. Concrete syntax definitions in GF assign concrete data types to the abstract categories, e.g. strings or record types, and provide implementations of the constructors as functions.

A concrete syntax for the above AST concerning railway object spacing is:

```
concrete RailwayEng of Railway = {
  lincat Object = Str; Length = Str;
         Restriction = Str; Statement = Str;
  lin Signal = ”signal”;
    Switch = ”switch”;
    Detector = ”detector”;
    LengthMeters i = i.s ++ ”m”;
    GreaterThan = ”greater than”;
    LessThan = ”less than”;
    ObjectSpacing o1 o2 r l = ”a” ++ o1 ++ ”must be” ++
        r ++ l ++ ”from a” ++ o2;
}
```

After both an abstract syntax and a corresponding concrete syntax has been defined, we can parse this language:

Text: *a switch must be more than 20 m from a signal*
AST:
```
ObjectSpacing Switch Signal
  LessThan (LengthMeters 20)
```

We can also linearize the language from the the abstract syntax:

AST:
```
ObjectSpacing Detector Signal
  LessThan (LengthMeters 2)
```
Text: *a detector must be less than 2 m from a signal*

Although this example is close to natural language, extending the language in the same style would quickly run into trouble trying to cover all the linguistic variation that arises from composing complex sentences. For example, adding words to the vocabulary which start with vowel sounds would require the article “a” to be replaced with “an” in these cases, breaking the compositionality of the program.

The resource grammar library defines a comprehensive set of linguistic categories such as noun phrases (`NP`), verb phrases (`VP`), clauses (`Cl`) and sentences (`S`) which can be used to compose texts. The type-safety enforced by the GF compiler on the constructors which use these linguistic categories ensures that the compositions are grammatical. Each language resource in the RGL implements these categories with the required attributes for that particular language. For example, the English grammar contains a determiner phrase `a_Det`, which can be linearized as “a” or “an”, depending on the composition of the noun phrase in which it is used. The example from above can be re-written to use the English resource grammar as follows:

```
concrete RailwayEngRGL of Railway = open SyntaxEng, ParadigmsEng,
    SymbolicEng, (Res=ResEng) in {
 lincat Object = N; Length = NP;
        Restriction = A2; Statement = S;
 lin Signal = mkN ”signal”;
  Switch = mkN ”switch”;
  Detector = mkN ”detector”;
  LengthMeters i = symb (i.s ++ ”m”);
  GreaterThan = mkA2 (mkA ”more”) (mkPrep ”than”);
  LessThan = mkA2 (mkA ”less”) (mkPrep ”than”);
  ObjectSpacing obj1 obj2 restriction length =
   mkS (mkCl (mkNP a_Det obj1)
    (mkVP (mkVP must_VV (mkVP (mkAP restriction length)))
    (SyntaxEng.mkAdv from_Prep (mkNP a_Det obj2))));
}
```

Using the resource grammar library allows us to separate the concerns of composing sentences from the concern of inflections and word ordering.

5.2.2 Design methodology overview

Our methodology is based on CNL and GF best practices; in particular, Ranta et al. [145] describe the construction of a CNL by creating an abstract syntax corresponding to a semantic model, mapping it into natural language, and also how to avoid or handle ambiguity in parsing and translating. In a later report, Ranta et al. [144] give explicit best practices, such as: (i) using a modular structure separating generic and domain-specific parts of the grammar, (ii) letting the AST model the semantics of the text, as opposed to the logic of the underlying formalism, and (iii) trade-offs in modelling language restrictions purely in context-free grammar versus using dependent types. We expand on these best practices as well as on the works from [82, 5, 31] that created domain-specific CNLs as verification front-ends.

We present here the methodology that we apply in Section 5.3 to design RailCNL, a verification front-end language for describing rules for static railway infrastructure verification. This methodology combines concrete advises from the above works with our own experience from creating a railway infrastructure verification platform (see Chapter 2).

The main activities for defining a verification front-end language using GF are:

1. Define an **abstract syntax** which is able to represent statements of relevant texts. We suggest two sub-activities to help manage the difficulty and complexity of modelling domain-specific, yet diverse and informally structured, texts:

   a) **Logic-driven design** where basic (often non-domain-specific) constructs that are known from the verification logic are added in a “bottom-up” fashion.

b) **Text-driven design** where highly domain-specific constructs are added to the language to model specific examples in original texts in a "top-down" fashion.

2. Write a **concrete syntax**, mapping the abstract syntax into one or more natural languages, using the Grammatical Framework and its resource grammar library.

3. Create a **translation** from the abstract syntax to the target logic formalism, i.e., the verification properties expressed in the input language of the solver.

In theory, these steps could be performed one after the other, each depending only on the previous steps in the list. In practice, however, the activities have subtle cross-dependencies, for example the need for reducing ambiguity by encoding more restrictions in the types, the usage of restricted keywords, or the need for structure on larger scales than a single sentence. Section 5.2.4 addresses each of these concerns.

Developing a specialized translation algorithm (see Section 5.3.2) instead of going through the GF typing system is encouraged when the end result is a complex logical language, as in our case. In the translation algorithm we can also incorporate various optimization aspects.

5.2.3 Abstract syntax

Attempting to formally model a body of informal specifications in its entirety may be neither feasible nor desirable, for a variety of reasons:

1. The text might have some amount of non-normative content intended only to give readers a better understanding of the subject matter.

2. Parts of the normative content might not be suitable for modelling in the target verification tool. For example, overly broad statements, such as "the system shall ensure safety in all possible conditions", are often part of regulations even if they do not lend themselves to any direct action. Our railway verification method is used for *static* infrastructure properties, whereas any properties requiring *dynamic* analysis are left to other stages (and tools) of the system design. A CNL can still be designed to model more properties than those which are translatable into the verification language.

3. The available body of text might be large and complex, and covering all parts of it could require diverse domain knowledge from various disciplines. In our railway case, we focus on the disciplines of track and signalling design, as these are the sub-disciplines of railway engineering for which we have had access to domain experts during the design of our verification system.

Furthermore, starting from arbitrary sentences in the natural text and trying to cover it with the CNL will often prove to be a daunting task, given the variety of

sentence structures, variety of contexts and levels of abstraction, and variety of domain knowledge needed to make sense of the statements.

Our approach to handling this difficulty is to split the process of designing the abstract syntax into two parts.

(a) We start with a *logic-driven design*, where we define basic concepts in a bottom-up fashion, such as classifying the statement types (*constraints*, *restrictions*, etc.) and describing sets of objects based on their class and their properties.

Even when deciding on the basic logic of the language, it might still be wise to abstract away from the details of the underlying verification logic (as noted in [144, Sec. 5.2]). In our railway verification case, even if many regulations can be concisely expressed as Datalog programs, the abstract syntax of these programs might not resemble the structure of the original text they were expressed in. As an example, Datalog does not nest predicates, so explicitly naming variables is required to express that an object has both a class and a property, while in natural language, a named variable would not be needed for such a statement.

Datalog: `main_signal(X) :- signal(X), type(X,main).`

By designing a language to have a level of abstraction closer to how the original texts are written, the details of the underlying formal language, its logic, or the verification system, might be changed without devaluing the knowledge base built by encoding domain knowledge into the front-end language. For example, the ontological statements in RailCNL could also be translated into an ontology language such as OWL.

(b) Next follows a *text-driven design* phase, where we look for text samples that can be captured in the CNL, and make adjustments and additions to the grammar to cover them. This phase might eventually lead to finding new basic building blocks, such as adding the *graph module* to RailCNL for describing railway layout, or adding *relations* to the ontology module. However, it is easy to get carried away and construct a highly nested language which has too much freedom and therefore becomes difficult to parse. Until the need for more generality is proven, each newly added construct is kept specific.

Alternating between the logic-driven and the text-driven phases can be useful for handling complexity and discovering the middle ground between informal specifications and verification logic. This approach follows the notion of *language oriented programming* [169], where identifying a high-level language to be used as a middle ground between bottom-up and top-down programming breaks the system design into two parts which can be handled separately. Similarly, we use the CNL as a middle-ground between the original texts and the verification system.

A consequence of this compromise is that the language will seldom be able to cover the exact wordings used in the original texts. We accept this consequence and aim instead to provide a user-friendly comparison of original text and CNL text for traceability (see Section 5.5.1). The logic-driven design phase is exemplified for our RailCNL in Section 5.3.1.

5.2.4 Concrete syntax

The abstract syntax is mapped into a natural language using the GF resource grammar library (RGL), which is well-covered in the GF documentation and literature (e.g. [145, 144]). Each category of the abstract syntax is mapped into a linearization type, often a record data structure. For example, the `Subject` category of RailCNL is assigned the complex noun (`CN`) record type, and `Statement` is assigned to utterance (`Utt`).

A major motivation for formal CNLs is that they can be unambiguously parsed as long as the language is restricted enough. Languages written using GF are often restricted to a pre-compiled vocabulary, to be able to identify structure and handle morphological variation. For our verification application, however, we need users to be able to *define new terms dynamically*, e.g. class names, and afterwards write statements using both built-in and user-defined terms. But allowing arbitrary string tokens can introduce ambiguity, i.e. the parser returning many parse trees for a single statement. We keep ambiguity under control through several means:

Type-level restrictions. The railway term “main signal” is the common way to refer to a signal which is of type *main*. A straight-forward way to add such modifying adjectives as a prefix to class names would be to add a constructor, for example:

```
Adjective : String -> Class -> Class
```

This constructor can be used to add adjectives to any class, and with the result of this operation also being a class. However, this approach would mean that any amount of arbitrary words ending with a class name (which could also be an arbitrary string) would be a valid parse. An undesired example is that the subject “a main signal which has height 10m” could be parsed as the class “10m” with six modifying prefix adjectives.

Usually, only one or two adjectives are prefixed to a class name. We can encode this restriction in the type system by separating the adjective-prefixed class name from the non-prefixed one. We also add two adjective constructors, one for adding an adjective and one for transforming the type `BaseClass` into `Class` without prefixing an adjective.

```
StringClassAdjective : String -> BaseClass -> Class
StringClassNoAdjective : BaseClass -> Class
```

In this way, we greatly reduce the ambiguity introduced by adding arbitrary prefixed strings to class names. We use this approach for RailCNL a number times, for example to add optional names to areas as described above.

Reserved keywords. Using arbitrary names as building blocks of our language resembles the use of identifiers as variables in programming languages. Programming languages have *restricted keywords* which cannot be used as variable names. Similarly, we use the GF parser callbacks system to remove parses which contain function words (such as "*which*", "*has*", "*is*", "*must*", "*be*", etc.) as arbitrary names. These are very unlikely to be needed as class or property names.

Weighted constructors. The GF parser has support for probabilistic grammars, which work by assigning weights (probabilities) to the constructors of the abstract syntax. By assigning a low weight to any constructor which uses the `String` category, we ensure that built-in syntax is always prioritized over arbitrary tokens.

Syntactic guides. As in programming languages, special symbols and punctuation can be used as guides for the parser if we are willing to compromise on naturalness. For example, in the following sentence we could use the curly brackets to indicate a class name (now allowing any number of tokens), and the square brackets could indicate placement.

Text: *A {home main signal} should not be placed [in a tunnel].*

For a human reader, if the meaning of the statement is preserved when ignoring the brackets, the CNL can still be said to be readable as natural text.

Alternatively, we can increase the verbosity of the syntax, to reduce the likelihood of causing ambiguity when embedded in a longer statement. Compare the following examples:

Text: *A signal of height 5.0m.*
Text: *A signal which has height which is equal to 5.0m.*

The second one is less likely to cause ambiguity when embedded in a longer statement. Adjusting the verbosity of the syntax is a method for making a trade-off between naturalness/conciseness and potential ambiguity.

5.2.5 Vocabulary: static versus dynamic

If the full vocabulary of the language is known in advance, we can define constant constructors which represent each atomic concept. In this case, the resource grammar library provides functions for setting up the required morphological variations

of lexical categories, for example by giving the stem and gender of a noun. However, we would like our CNL to be able to also define new terms and subsequently construct statements using these new terms. This would imply that the vocabulary is not static, and cannot be compiled in advance.

The most important lexical category for dynamic vocabulary would be nouns, possibly composite nouns. Most of the morphological variation for these (in Norwegian) would be given by their gender. A work-around for dynamic vocabulary could be to encode the gender in the abstract syntax. This would allow natural and consistent use of gender for noun added dynamically to the vocabulary, but this technique ties the AST to the concrete language and could thus make it harder to handle several concrete syntaxes.

To avoid excessive ambiguity caused by allowing arbitrary words in the grammar, we can declare a set of *keywords* which should never be parsed in the arbitrary names category. This is implemented in the Grammatical Framework's runtime library as a callback function which disqualifies certain parses by examining the arbitrary word.

Another work-around used in [82] is to write new vocabulary items back into the GF source code for the language and recompile the GF grammar. We rule this approach out for this project to avoid having to distribute the GF compiler and Haskell runtime with our CAD tool (see Section 5.5).

5.2.6 Translation into the target logic formalism

If the abstract syntax is made to faithfully model the logic of the verification system, then the translation into the logic formalism can be made by implementing another GF concrete syntax for the target language. However, target logics are often too low-level to represent regulations directly. GF incorporates dependent type features which could allow for a more concise representation of this translation, but this practice has not yet matured to a state in which it can be said to be a recommended practice (see [144]). For RailCNL we have instead written a separate program (in C#, as it is a part of the verification CAD plugin) which translates from the abstract syntax of the CNL into Datalog. Section 5.3.2 describes the main techniques that we used for RailCNL.

5.3 RailCNL: a front-end language for railway verification

With RailCNL we aim to cover the following content (see Section 5.4 on page 119 for a detailed account of the coverage that we achieve):

1. Definitions of railway-domain terms from a set of basic terms given by the object types present in the CAD program and the railML exchange format.

```
⟨Statement⟩ ::= ⟨OntologyAssertion⟩
  | ⟨OntologyRestriction⟩
  | ⟨DistanceRestriction⟩
  | ⟨PathRestriction⟩
  | ⟨PlacementRestriction⟩
  | (...) // Partial grammar

⟨OntologyAssertion⟩ ::= ⟨Subject⟩
    ⟨Condition⟩

⟨OntologyRestriction⟩ ::= ⟨Subject⟩
    ⟨Modality⟩ ⟨Condition⟩

⟨DistanceRestriction⟩ ::=
    'the distance from' ⟨Subject⟩
    'to' ⟨GoalObject⟩ ⟨Modality⟩
    ⟨Restriction⟩

⟨PathRestriction⟩ ::= ⟨PathQuantifier⟩
    'from' ⟨Subject⟩ 'to' ⟨GoalObject⟩
    ⟨Modality⟩ ⟨PathCondition⟩

⟨PlacementRestriction⟩ ::= ⟨Subject⟩
    ⟨Modality⟩ 'be placed in' ⟨Area⟩

⟨Modality⟩ ::= 'must' | 'shall not'
  | 'should' | 'should not'

⟨PathQuantifier⟩ ::= 'all paths'
  | 'no paths' | (...)

⟨PathCondition⟩ ::= 'pass'
    ⟨DirectionalObject⟩

⟨GoalObject⟩ ::= ⟨DirectionalObject⟩
  | 'the first' ⟨DirectionalObject⟩

⟨DirectionalObject⟩ ::= ⟨SearchSubject⟩
  | 'a facing switch'
  | 'a trailing switch'
  | ⟨SearchSubject⟩ ⟨RelativeDirection⟩

⟨RelativeDirection⟩ ::= 'same dir.'
  | 'opposite dir.'

⟨SearchSubject⟩ ::= 'a' ⟨Subject⟩
  | 'another'

⟨Area⟩ ::= ⟨BaseArea⟩
  | ⟨BaseArea⟩ 'which has'
    ⟨PropertyRestriction⟩
  | ⟨Area⟩ 'or' ⟨Area⟩
  | ⟨Area⟩ 'and' ⟨Area⟩

⟨BaseArea⟩ ::= 'tunnel' | 'bridge'
  | 'local release area' | ⟨Identifier⟩

⟨Subject⟩ ::= 'a' ⟨Class⟩
  | 'a' ⟨Class⟩ 'which' ⟨Condition⟩

⟨Condition⟩ ::= 'is a' ⟨ClassRestriction⟩
  | 'has' ⟨PropertyRestriction⟩
  | 'is a' ⟨ClassRestriction⟩
    'which has' ⟨PropertyRestriction⟩

⟨PropertyRestriction⟩ ::= ⟨Property⟩
    ⟨ValueRestriction⟩
  | (...) // and/or

⟨ClassRestriction⟩ ::= ⟨Class⟩
  | (...) // and/or

⟨ValueRestriction⟩ ::= ⟨Value⟩
  | 'not equal to' ⟨Value⟩
  | 'less than' ⟨Value⟩
  | (...) // ≤, >, ≥
  | (...) // and/or

⟨Value⟩ ::= ⟨Identifier⟩ | ⟨Number⟩
    ⟨Unit⟩

⟨Property⟩ ::= ⟨Identifier⟩

⟨Class⟩ ::= ⟨Identifier⟩
```

Figure 5.2: English version of RailCNL's core grammar in BNF (GF notation shown in Appendix 5.7). Some linguistic complexity such as subject-verb agreement is ignored here; the actual grammar is fully specified as GF code, which is ideally suited for handling such cases.

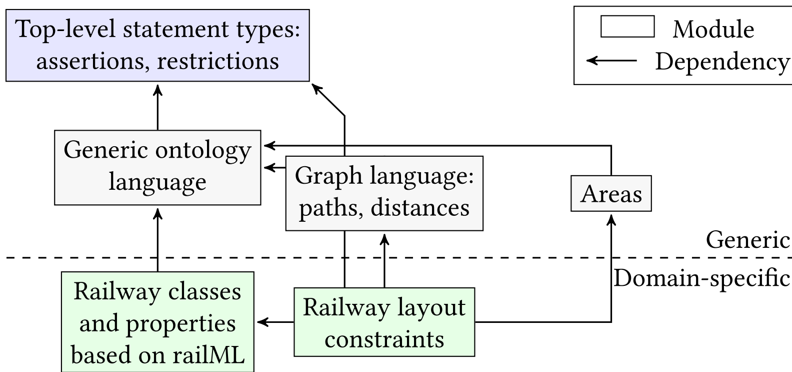


Figure 5.3: Modules of the RailCNL (boxes) and their dependencies (arrows). The *generic* modules could be reused when building CNLs for verification in other domains. The *specific* modules are, however, tailored to railway regulations.

2. Regulations (from infrastructure manager technical regulations[4]) which give obligations or recommendations on the design of the railway infrastructure.

3. Expert knowledge given in textual form apart from official regulations, used to gather and formalize engineering practice.

An English version of RailCNL's core grammar is presented in Figure 5.2. The full grammar is defined in GF (see Appendix 5.7 at page 134 for an excerpt of the RailCNL grammar in GF notation), which has some advantages over classical BNF parsers: (i) separation of abstract syntax and concrete syntax; (ii) resource grammar library for natural languages, allowing us to compose sentences in natural language while abstracting away from morphological details; (iii) modularity and extensibility, which we need for evolving a domain-specific language alongside its application; and (iv) tool support for managing text (editors, predictive parsing, visualization).

RailCNL has been developed to support Norwegian language regulations. All the examples presented below have been translated into English for the purpose of presenting them here.

5.3.1 RailCNL modules and examples

RailCNL has a modular design (see Figure 5.3) where domain-specific constructs are separated from generic ones. However, CNL modules are not always trivially composable, and care must be taken to retain naturalness while avoiding ambiguity

[4]Norwegian infrastructure manager Bane NOR's regulations: `https://trv.jbv.no/`

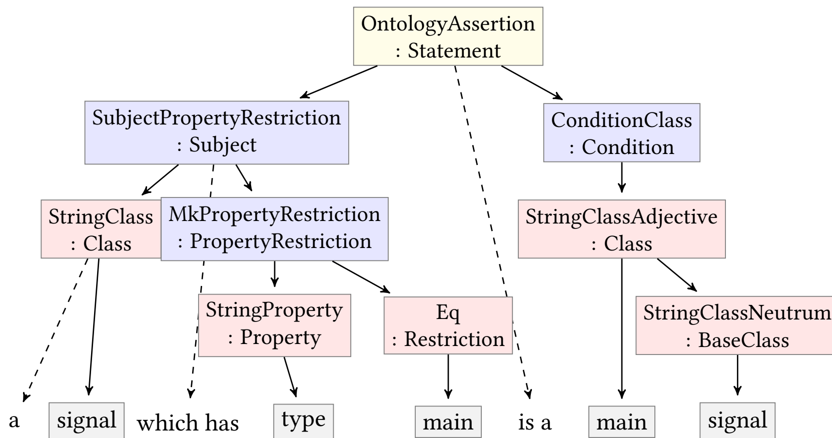


Figure 5.4: The parse tree of the ontology assertion statement from Example 1.

when increasing the complexity of the language (as presented in Section 5.2). We describe below the main modules and constructs of RailCNL, with examples of CNL text and the corresponding abstract syntax tree (AST) obtained from the GF parser.

5.3.1.1 Top-level statement types

Most normative sentences in railway regulations are classified into one of the following types, or their negation:

- **Constraint**: logical constraints on the railway infrastructure model. These sentences can be used by the Datalog reasoner to infer new statements.

Example 1 (Parse tree for a constraint statement).
CNL: A signal which has type main is a main signal.
AST:

```
OntologyAssertion
  (SubjectPropertyRestriction
    (StringClass "signal")
    (MkPropertyRestriction (StringProperty "type")
                           (Eq (MkValue
                                  (StringTerm "main")))))
  (ConditionClass (StringClass "main_signal"))
```

- **Obligation**: design requirements on the railway infrastructure. The CAD model is checked for compliance, and violations are presented as errors to the user.

Example 2 (Parse tree for an obligation statement).
CNL: A vertical segment must have length greater than 20.
AST:

```
OntologyRestriction Obligation
  (SubjectClass (StringClassGen1 "vertical segment"))
  (ConditionPropertyRestriction
    (MkPropertyRestriction (StringProperty "length")
                           (Gt (MkValue (StringTerm "20")))))
```

- **Recommendation**: design heuristics for railway infrastructure. The CAD model is checked for compliance, but violations are presented as warnings or for information only, which can be dismissed from the view.

Example 3 (Parse tree for a recommendation stmt.).
CNL: A switch should be placed on a straight segment.
AST:

```
PlacementRestriction Recommendation
  (SubjectClass (StringClass "switch"))
  (SingleArea (NoRestrictionArea
                (NonSpecificArea
                  (MkNamedArea "straight segment"))))
```

5.3.1.2 Generic ontology module

Statements about classes of objects and their properties form a natural basis for knowledge representation. We allow arbitrary string tokens to represent class names, property names and values, and compose these in various ways.

- **Class names**: are arbitrary words, optionally prefixed with another arbitrary word. The reason for allowing this is to give the CNL the power to define new words. As an example, we define “railway object”:

Example 4 (Parse tree for using class names).
CNL: A signal is a railway object.
AST:

```
OntologyAssertion
  (SubjectClass (StringClassNoAdjective
                  (StringClassNeutrum ”signal”)))
  (ConditionClassRestriction
    (MkClassRestriction (StringClassAdjective ”railway”
                          (StringClassNeutrum ”object”))))
```

- **Properties and values**: can be arbitrary string tokens. These can be joined by “and” or “or” both on the level of values and of properties.

Example 5 (Parse tree using properties and values).
CNL: A project which is a new construction should have quality normal or high.
AST:

```
OntologyRestriction Recommendation
  (SubjectCondition (StringClassNoAdjective
                      (StringClassNeutrum ”project”))
    (ConditionClassRestriction
      (MkClassRestriction
        (StringClassAdjective
          ”new” (StringClassNeutrum ”construction”)))))
  (ConditionPropertyRestriction
    (MkPropertyRestriction
      (StringProperty ”quality”)
      (OrRestr (Eq (MkValue (StringTerm ”normal”)))
               (Eq (MkValue (StringTerm ”high”))))))
```

- **Restrictions**: Equality (shown as `Eq` in the AST example above) is a common case of restriction for which we simply choose the wording “to be”. Other restriction types such as greater than (`Gt`), less than (`Lt`), etc. are worded more verbosely.

Example 6 (Parse tree using restrictions).
CNL: A main signal should have height which is greater than 1.5m and less than 5.0m.
AST:

```
OntologyRestriction Recommendation
  (SubjectClass (StringClassAdjective
                 "main" (StringClassNeutrum "signal")))
  (ConditionPropertyRestriction (MkPropertyRestriction
     (StringProperty "height")
     (AndRestr (Gt (MkValue (StringTerm "1.5m")))
               (Lt (MkValue (StringTerm "5.0m"))))))
```

- **Relations**: the basic ontology module contains multiplicity restrictions on relations. In the layout module presented below, we will see how relations are used when writing statements which are concerned with more than one object simultaneously.

Example 7 (Parse tree using relations).
CNL: A distant signal should have one or more associated signals.
AST:

```
OntologyRestriction Obligation
  (SubjectClass (StringClassAdjective
                 "distant" (StringClassNeutrum "signal")))
  (ConditionRelationRestriction ManyRelation
    (StringClassAdjective "associated"
      (StringClassMasculine "signals")))
```

5.3.1.3 Layout module

For writing statements about the topology of the railway track, e.g. about paths as illustrated in Figure 5.5c, we use the following language constructs:

- **Goal object**: modifies the `Subject` type defined in the ontology module to add conditions which make sense in a railway graph search, such as the object's orientation (same direction or opposite direction) the search's direction (forwards or backwards) or the termination properties of the search.
- **Path condition**: argument to the search constructors which specifies what restrictions are placed on the paths from source to goal object.
- **Path restrictions**: the combination of the source object, goal object and path conditions. (See Figure 5.5a)

Example 8 (Parse tree using path restriction).
CNL: All paths from a station border to the first facing switch must pass an entry signal.
AST:

```
AllPathsObligation (SubjectClass
    (StringClassAdjective ”station”
                          (StringClassMasculine ”border”)))
  (FirstFound FacingSwitch)
  (PathContains (AnyDirectionObject (AnySearchSubject
      (SubjectClass (StringClassAdjective ”entry”
                          (StringClassNeutrum ”signal”))))))
```

- **Distance restrictions**: See also Figure 5.5b.

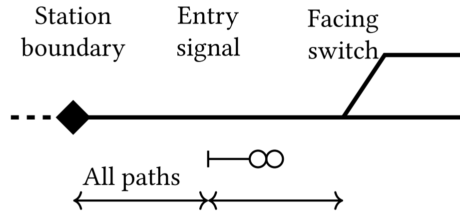
Example 9 (Parse tree using distance restriction).
CNL: The distance from an entry signal to the first facing switch must be greater than 200m.
AST:

```
DistanceObligation (SubjectClass (StringClassAdjective
                  ”entry” (StringClassNeutrum ”signal”)))
  (FirstFound FacingSwitch) (Gt (MkValue (StringTerm ”200m”)))
```

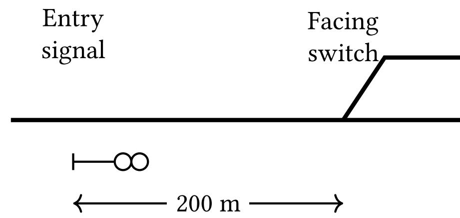
5.3.1.4 Area module

The area module modifies subjects to express whether they are inside a planar area, such as station areas, tunnels or bridges, or belongs to a linear segment of a track, such as being located in a curve or on an incline (see Figure 5.5d).

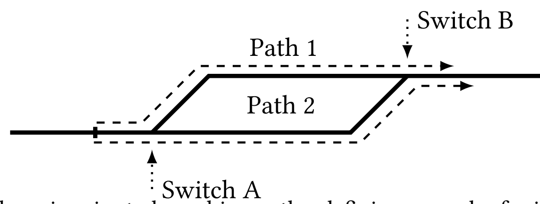
- **Subject constructor**: the `Subject` is extended to add a prepositional phrase containing area information, such as being inside of a tunnel or on a bridge.
- **Placement restriction**: extends the constructors for the type `OntologyRestriction` to allow restrictions on object being inside areas.



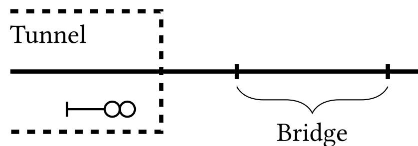
(a) Path restrictions are constructed from a subject, a goal, a quantifier and a condition.



(b) Distance restrictions are constructed from a subject, a goal, and a value restriction.



(c) Switches give rise to branching paths, defining a graph of railway tracks.



(d) Area containment can refer to either a planar region or an interval on a track.

Figure 5.5: Conditions on railway geographical layout as supported by RailCNL.

Example 10 (Parse tree using areas).
CNL: *A main signal should not be placed in tunnel or bridge.*
AST:

```
PlacementRestriction NegativeRecommendation
  (StringClassAdjective "main" (StringClassNeutrum "signal"))
  (MkArea (OrArea (SingleAreaConj (NoRestrictionArea
                    (NonSpecificArea TunnelArea)))
                  (SingleAreaConj (NoRestrictionArea
                    (NonSpecificArea BridgeArea)))))
```

5.3.1.5 Signalling layout regulations

- **Running times**: a variation on the distance restriction is to use running time (travel time) from one place to another. These are used as heuristics for the control system's performance. This running time can be, e.g., *nominal speed* (allowable speed) or *maximum dynamic time* (maximum speeds taking acceleration and braking into account).

Example 11 (Heuristic for axle counter placement.).
CNL: *Running time at nominal speed from an axle counter to an adjacent axle counter must be less than 25s.*
AST:

```
RunningTimeObligation NominalSpeed
  (SubjectClass (StringClassAdjective "axle"
      (StringClassMasculine "counter")))
  (AnyFound (AnyDirectionObject SubjectOtherImplied))
  (Lt (MkValue (StringTerm "25s")))
```

5.3.2 Translating RailCNL into Datalog

To make use of RailCNL in the verification tool, ASTs obtained by parsing CNL phrases with the GF runtime are transformed into Datalog rules. Each top-level constructor in the CNL definition has a translation function into the Datalog AST.

Predicate conventions. We employ the following predicate conventions:

- Class membership as `classname(object)`.
- Object properties as `propertyname(object, value)`.
- Relation between objects as `relationname(object,otherobject)`.

Explicit variables. The *Subject* of the sentences of the *Ontology* module defines an arbitrary individual whose definition does not depend on other information. To translate it, we create a new variable denoting the arbitrary individual.

Example 12 (Datalog translation of a subject constructor).
CNL: *A signal which has height 4.5m*
Datalog: `signal(X), height(X, 4.5).`

The subject is the starting point for the translation, as other parts of the phrase refer back to the subject. In the following example, we first process the `SubjectCondition` part of the sentence ("A signal which has height 4.5m"), find a fresh variable name for it, and then process the consequent ("is a tall signal"), which implicitly refers to the subject ("X").

Example 13 (Datalog translation using implicit variable reference).
CNL: *A signal which has height 4.5m is a tall signal.*
AST:

```
OntologyAssertion (SubjectCondition
    (StringClassNoAdjective (StringClassNeutrum "signal"))
    (ConditionPropertyRestriction
      (MkPropertyRestriction
        (StringProperty "height")
        (Eq (MkValue (StringTerm "4.5m"))))))
  (ConditionClassRestriction
    (MkClassRestriction (StringClassAdjective "tall"
                         (StringClassNeutrum "signal"))))
```

Datalog: `tall_signal(X) :- signal(X), height(X, 4.5).`

Ontology assertions. As seen in the previous example, translations of ontology assertions take the subject, construct a rule body from it, then take the consequent condition, and create a rule head containing a rule head from it. As Datalog allows only single predicates as rule heads, this means that we cannot write assertions which imply disjunctions. For example, the following text can be parsed by our CNL parser, but not translated to Datalog.

Example 14 (Ontology assertion which would result in disjuntive head and is thus not expressible in Datalog).
CNL: A signal has height 4.0m or 4.5m.
AST:

```
OntologyAssertion (SubjectClass (StringClassNoAdjective
     (StringClassNeutrum ”signal”)))
 (ConditionPropertyRestriction (MkPropertyRestriction
     (StringProperty ”height”)
     (OrRestr (Eq (MkValue (StringTerm ”4.0m”)))
              (Eq (MkValue (StringTerm ”4.5m”))))))
```

This limitation corresponds to the theoretical restrictions on Datalog. Allowing such sentences is the defining characteristic of a Datalog extension called *Datalog with disjunctive heads*, which has higher computational complexity than plain Datalog. For example, three-colouring of a graph would be expressible in Datalog with disjunctive heads. Note that merely checking that all signals have height either 4.0m or 4.5m is certainly expressible in Datalog, and is covered by ontology restrictions.

Ontology restrictions. For ontology restrictions, such as obligations (“must”) and recommendations (“should”), the Datalog rule head contains a predicate which captures any violations of the text. This is achieved by first defining the restrictions themselves (`r1_found` in Example 12 below) and then declaring a rule which uses the negation of these restrictions (`!r1_found`) in order to yield a counter-example.

Example 15 (Datalog translation of an ontology restriction).
CNL: A signal must have height 4.0m or 4.5m.
AST:

```
OntologyRestriction Obligation
  (SubjectClass (StringClassNoAdjective
      (StringClass ”signal”)))
  (ConditionPropertyRestriction
    (MkPropertyRestriction (StringProperty ”height”)
      (OrRestr (Eq (MkValue (StringTerm ”4.0m”)))
               (Eq (MkValue (StringTerm ”4.5m”))))))
```

Datalog:

```
r1_found(Subj0) :- signal(Subj0), height(Subj0, 4.0).
r1_found(Subj0) :- signal(Subj0), height(Subj0, 4.5).
r1_obl(Subj0) :- signal(Subj0), !r1_found(Subj0).
```

Disjunctive normal form. As Datalog does not necessarily have an *or* operator, nor negation over complex terms, these must be factored out into separate rules and auxiliary predicates. This transformation can be performed by considering the result of the translation of a sentence to be a *set of rules* (such as the two definitions of `r1_found` in Example 12), and the result of the partial translation (such as adding a class or property constraint to a rule) to be a *set of conjunctions* which are prefixes of the final rules.

Vocabulary matching. The Norwegian regulations are written in Norwegian and use other terms for class names, properties and relations than the railML representation does. After identifying the class names from the CNL, they will be looked up in a Norwegian/railML dictionary. For example, Norwegian "*akselteller*" is mapped into the railML class "*trainDetector*" with the "*axlecounting*" property.

Simplifications and optimizations. Creating Datalog rules for **layout properties** requires reasoning about paths and distances of a directed graph. We start from a relation describing edges of the graph, from e_1 to e_2 with distance d is $\text{next}(e_1, e_2, d)$. It could be possible to define general connected and distance predicates, as we have used in Chapter 2. However, this can become inefficient, especially if using a *bottom-up materializing* Datalog solver, which would then compute the transitive closure of the whole graph and distances of all paths in the graph. For a single design concerning a small to medium-sized train station, this might be acceptable as verification procedure, but to achieve our goal of on-the-fly verification and large-scale verification of railway lines spanning many stations, or even a whole national network, we must ensure that rules can be *localized*. For example, specifying minimum distances between objects (such as the minimum separation of 21.0m for train detectors) should not lead to calculation of distances between all pairs of train detectors.

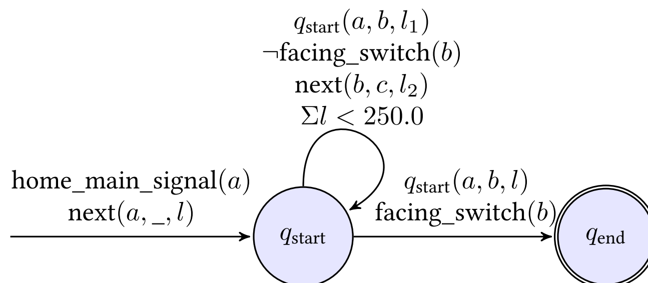


Figure 5.6: Datalog rules used to execute the distance search from "home main signal" to "first facing switch".

To achieve a local search, we avoid the global distance and connected predicates, and use instead the underlying next relation directly when translating the CNL to Datalog. We think of the search as a state machine with one Datalog rule corresponding to each state, see Figure 5.6. One or more searching states are recursively defined to create a transitive closure of the next relation, guarded by distance conditions, path conditions, etc. Finding the search goal, under the given conditions, leads to an *accepting state*, which is a relation containing the violation of the specification given in the text.

Example 16 (Datalog distance search).
CNL: *The distance from a home main signal to the first facing switch must be greater than 250.0m.*
AST:

```
DistanceRestriction Obligation (SubjectClass
    (StringClassNoAdjective (StringClassNeutrum
                              ”home_main_signal”)))
  (FirstFound FacingSwitch)
  (Gt (MkValue (StringTerm ”250.0m”)))
```

Datalog:

```
r1_goal(Goal0)    :- switch(Goal0).
r1_start(S0,E,D) :- signal(S0), next(S0, E, D).
r1_start(S0,E,D) :- r1_start(S0, M, D0), next(M, E, D1),
                    D=D0+D1, D < 250.0, !rule1_goal(M).
r1_end(S,E,D)    :- r1_start(S,E,D), rule1_goals(E).
```

Inlining. Simple instances of `OntologyRestriction` statements can often be written as a single rule. However, the general translation procedure splits this up into finding correct instances (predicate name ending in “`found`”), and a separate rule identifying the same objects with a negation of the `found` rule (predicate name ending in “`obligation`” or “`recommendation`”). Whenever the `found` predicate has only a single atom which is different from the obligation/recommendation rule, then it can be inlined into the same rule.

Example 17 (Inlining).
CNL: A sign should have angle to the track tangent which is greater than 94°.
Datalog:
```
sign_found(Subj0) :- sign(Subj0), tangent(Subj0, Val2),
                     Val2 > 94.
sign_recommendation(Subj0) :- sign(Subj0), !sign2_found(Subj0).
```
After performing inlining simplification we get instead:
Datalog:
```
sign_recommendation(Subj0) :- sign(Subj0), tangent(Subj0, Val2),
                              Val2 >= 94.
```

5.4 Evaluation of RailCNL coverage of Norwegian regulations

Table 5.1 is based on an analysis of phrases from a selection of technical regulations from Norwegian infrastructure manager Bane Nor (`https.//trv.banenor.no/`). Content from regulations concerning the engineering disciplines of railway tracks and signalling were selected for the evaluation because they were the focus of the RailCOMPLETE development, and as such was the domains for which the company had available expertise. (See Appendix 5.8 at page 135 for representative examples and Appendix 5.9 for a comprehensive overview of the Norwegian technical regulations that we worked with.)

Each sentence or table cell of the original text was classified according to the following:

1. **Sentence type**:classifying the sentence into *formalities* (headings, captions, etc), *meta* (describing the text), *applicability* (declaring scope, referring to other sections, etc.), *normative* (considered relevant for translation to RailCNL), or *other*.

2. **Discipline**: identifying whether phrases were belonging to another discipline than what the chapter heading had declared.

3. **Stage**: identifying whether the phrase was relevant for the planning stage of a railway construction project, excluding e.g. generic construction or operation statements.

4. **Static checkability**: for normative statements relevant for planning, there is also the possibility that they do not fit into the layout and specification part of the planning, thus not being suitable for static infrastructure verification. This is most notably the case for railway interlocking (control system) regulations, where statements about the dynamic behaviour of the control system (concerning e.g. latency, timeouts, and state) are typically not part of the station-specific specification, and are not relevant for static infrastructure verification.

Eng. disc.	**Chapter title**	Phrases	Normative	Relevant	Covered	**Coverage**
Track	Planning: general techn.	140	74	74	70	95%
Track	Planning: geometry	278	157	152	119	78%
Signalling	Planning: detectors	144	106	35	21	60%
Signalling	Planning: interlocking	376	265	130	81	62%
Total		938	602	391	291	**74%**

Table 5.1: Coverage evaluation for a subset of Norwegian regulations. *Phrases* of the original text which could be classified as *normative* (i.e. applying some restriction on design) were evaluated for *relevance* to static infrastructure verification. The *coverage* is the percentage of relevant phrases expressible in RailCNL.

Table cells from the regulations were considered separate phrases, e.g. a number in a table cell was re-phrased as a self-contained CNL statement using information from the row and column headers. Phrases that were reasonably naturally expressible in RailCNL (either straight-forwardly in the basic logic, or after adding appropriate domain-specific constructs), were counted as *covered.* The results are detailed in Table 5.1.

5.5 Tool integration

There are three ways of making use of a CNL for participatory verification.

Reading is the most simple to implement, but offers least benefits. For participatory verification purposes, only allowing an engineer to read in natural language the verification properties that the verifier works with is already valuable as it establishes trust in the opaque verification mechanisms. Moreover, using the CNL would shield the engineer from various logical formalisms that are used by complex processes like formal verification or certification. In Section 5.5.1 we show how using RailCNL only in a reading mode allows us to provide real help to the engineer in understanding the errors reported by our verification engine in a way that the engineer can make sense out of.

Template Editing, as a limited tool-support for editing phrases, would offer simple forms of editing, in addition to all the reading benefits. Changing a numerical value or changing words by selecting from a list of choices is easy to do and easy to understand for users, without requiring full understanding of the formal grammar behind the structure of the phrase. In the railway domain, the national regulations change seldom, and when they do, it is often enough to do only simple changes, e.g., when a new speed limit is imposed we only need to change a number. Besides being useful to regulators, template editing can also be used by engineers in companies to adjust regulations or properties that their specific designs need (maybe only temporarily, like for debugging purposes).

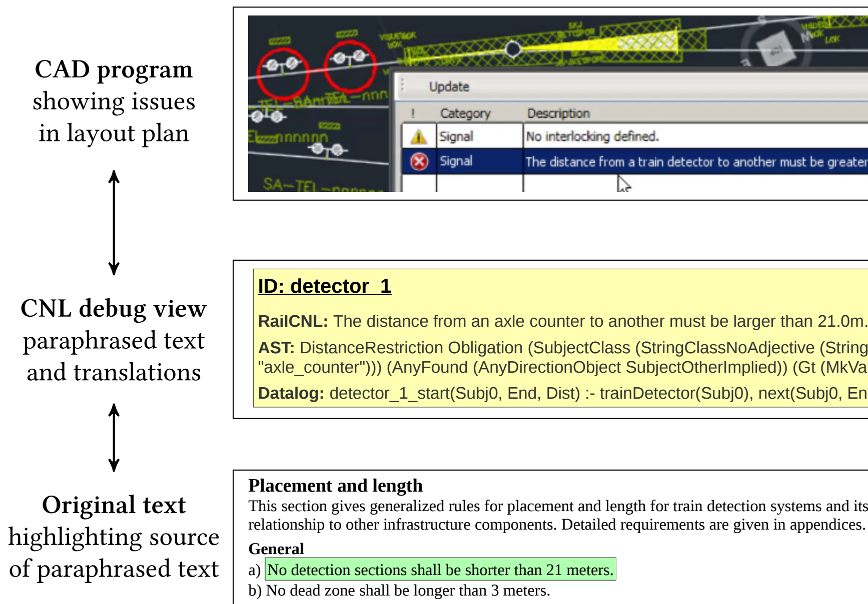


Figure 5.7: Tracing of requirements backwards from CAD program through CNL to marked-up original texts. From a regulation violation presented as a warning or error, the user can browse to the corresponding regulatory text, shown side by side with the CNL text.

Writing and editing phrases allows the CNL to be used at its full potential, but developing an editor which allows a user to edit phrases without having a thorough understanding of the formal CNL grammar is a challenge. Section 5.5.2 presents the many difficulties and features of CNL editors. Model-checking is not only useful for verifying compliance with regulations, but more importantly engineers would like to add 'rule-of-thumb' properties which they normally abide by when making new designs. These are usually considered valuable and proprietary for a company. The RailCNL editor that we present in Section 5.5.3 is meant to allow railway engineers to write properties in their natural domain language which can be verified by our engine, so that their know-how is kept in-house, without the need for an external expert (in verification).

5.5.1 Traceability support in RailCNL

Verification tools usually output a counter-example when the requirements are violated by the model. It is often difficult to understand from the counter-example

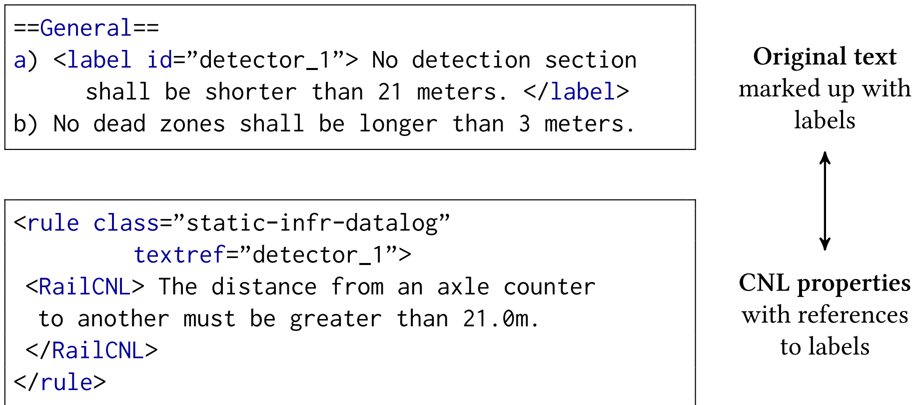


Figure 5.8: Excerpt of original text marked-up with sentence identifiers, and properties represented in CNL with references to original text.

which of the (possibly several) requirements have been violated, and why. We use the notion of *tracing* to trace such errors from the verification output all the way to the original text regulations. Figure 5.7(top) shows our prototype tool (running as a plug-in for the AutoCAD program used by Norwegian railway engineers) presenting a problem in the CAD view. Figure 5.7(middle) shows how the error message can be traced back through the Datalog code, the AST, and the CNL code, to the original, highlighted, regulations text Figure 5.7(bottom).

We mark-up sentences of the original text with an identifier, and create a separate document containing the formalized representation using RailCNL, using the identifiers as references back into the original text (Figure 5.8). When the verification program finds a violation among the regulations, it outputs the identifier of the rule that has been violated, enabling the tracing.

When producing new regulatory texts or writing down expert knowledge for which a CNL exists, the approach of *embedded controlled language* [142] can be used to create natural texts where some sentences are directly parsable into verification properties. The other parts of such a text is then considered to be comments or explanations, similar to the programming approach known as *literate programming*.

5.5.2 An overview of CNL editors and their features and properties

While a CNL is usually designed to be easy to read without any prior training, the process of *writing* in a restricted language is less straightforward. In order to write

in CNL, users need to be able to construct phrases which are correct with respect to the particular syntactic restrictions of that language, and which moreover have the intended meaning.

A CNL that presents itself as user-friendly may not expect that its users are willing to study its rules thoroughly before attempting to compose something in that language. This poses a challenge: the user wants to compose a phrase which is structurally correct, yet without having to know the rules governing that structure, or even seeing the underlying structure at all. This challenge is often aided by some software interface or tool designed specifically to help the user construct phrases which are valid in the particular CNL. We refer to such tools generally as *CNL editors*.

In this section, we give an overview of the state of the art in this area and compare the features of various CNL editors found in the literature. This section is useful to understand the choices that we made when developing the RailCNL editor presented in Section 5.5.3.

When we talk about CNL editors, we are implicitly assuming that the goal of the CNL in question is formalization, and thus that there exists some formal definition of the CNL itself, along with a parser. Therefore, we will not consider CNLs that have not been formalized or for which no parser exists.

There are two predominant paradigms to CNL editing:

- **Structural editing**: where the user is building a formal representation (generally a tree) in a structural way, prevented from going outside the bounds of the CNL.
- **Surface editing**: where the user is inputting text, with varying degrees of guidance from the editor, which eventually needs to be checked for conformity.

These can be seen as the opposite ends of a spectrum where most CNL editors out there can be placed neatly at either end, while only a few fall in between.

This distinction goes back to [170], who outline the design issues that arise in the construction of language-based editors. Even though [170] treats primarily programming languages and not CNLs, many of the ideas discussed there are quite relevant to CNL editing:

> *"The assumption that all users are willing and able to think exclusively in tree terms is clearly false. In practice, many users have a pluralistic view of the programs they manipulate, seeing them sometimes as tree structures, sometimes as symbol sequences, and sometimes as character texts."*

The authors put forward the idea of *pluralistic editors*, which should support editing on these different levels of abstraction. These ideas are not surprising, and seem to serve as the basis for most of the CNL editors we have seen.

5.5.2.1 Structural editing

The idea of structural editing for programming languages goes back at least to Mentor [48] and the Cornell Program Synthesizer [162]. These early editors were based on the philosophy that since programs are not text, it does not make sense to treat them as such. *"They are hierarchical compositions of computational structures and should should be edited, executed, and debugged in an environment that consistently acknowledges and reinforces this viewpoint."*[162]. These programming tools put the underlying structure of the language very much in focus, minimizing the role of their text-based concrete syntax. This demands full understanding by the user of the language's grammar. While this idea did not really catch on for mainstream programming languages (which are still very much written and debugged using text-based formats) it still comes up in a number of CNL editors, in various guises. Scott and Power [136] introduce the idea of WYSIWYM editing (What You See Is What You Meant) for multilingual authoring, where users edit on the level of semantic representation, in their case a knowledge base, which is linearized into "feedback texts" in multiple languages for the user during the editing phase.

This idea is also mirrored in the syntax tree editing tools for CNLs defined using the Grammatical Framework [141] of which there are a few different implementations [90, 117, 29]. In these editors the user is directly building a tree in a top-down fashion by choosing the functions to be used at each node. The partial tree can be immediately linearized in multiple languages while the user works, including holes for yet unspecified nodes. Sub-trees can be inserted by parsing free text, but no guidance is provided to the user at this stage; the parse either succeeds or fails, and ambiguities must be fixed manually.

Ljunglöf [101] proposes the use of interactive tree building for dialogue management. The general idea is that a tree is built by asking questions to the user and then the tree is refined based on the responses obtained, filling in the missing holes until a complete tree is obtained. There is however no concrete tool associated with this work.

5.5.2.2 Structured surface editing

This class of editors comes somewhere in between the two major extremes described above. They are *structural* in the sense that you are directly building a tree and cannot stray outside the CNL, but they are also *surface-based* because the user is interacting with the editor on the surface string level, not necessarily seeing the underlying structure at all.

5.5.2.2.1 **Menu-based input**. Within this class, a number of editors tend to use similar user interface components, in particular the ideas of templates with holes which are filled by selecting completions from some kind of menu. An early example of this idea can be found in [163], which continued into various other works

including NLMenu [164], ROSY [17], and MenuGen [69], which all follow more or less the same ideas. The authors of [167] discuss a general architecture of a menu-based input system for a multilingual translation systems, underlining the benefits of menu-based input for avoiding user input error.

The authors of [68] coined the term *Conceptual Authoring* for their input system for building database queries. Continuing this menu-based idea, their interface replaces the traditional query-writing as plain text with a set of templates with holes which are filled using menus. All editing operations are defined directly on an underlying logical representation, governed by a predefined ontology. The method avoids the problems associated with parsing, and is particularly well suited for query interfaces to closed-domain systems.

Similarly, the Phrasomatic editor[5] is a web-based interface for multilingual CNL phrases, powered by GF grammars. It has a distinctly menu-driven approach where the kinds of phrases one can write are pre-determined, and the user's task is to fill slots from the existing lexicon. The interface is hard-coded to be grammar-specific, in the sense that it is not generated by the grammar alone. This approach makes sense for small languages where the number of possible choices at each point is small, and there is no benefit to be had by allowing free-text input.

A slightly different approach within this class of editors is the MUSTE editor[6] [102], which is an experiment in keyboard-free structural editing on the surface level, where the tree is not revealed to the user. There is no text input at all, instead, the user edits an existing phrase by clicking words to show possible replacements. Clicking multiple times changes the scope currently under focus, extending from words to sub-phrases, and allowing potentially any sub-tree to be edited in accordance with the underlying grammar. It is an experiment in editing techniques and does not scale to larger CNLs.

5.5.2.2.2 **Blocks and frames**. Some editors are based on the concept of blocks which fit together and reveal the underlying structure of the text. The Blockly project[7] provides a UI library for building such kinds of interfaces. It is mainly promoted as an educational tool for teaching people to understand the structure behind programming languages. Colour-coding and visual connectors are used to give some type information, for example distinguishing between statements and expressions. Alice[8] is a programming environment for creating animations and program simple games in 3D, which uses an editor based on this block-based interface.

In a similar fashion, the ATTAC-L editor [166, 44] uses "bricks" as the basic building blocks of its language, which can be pieced together and which reveal

[5] `http://www.phrasomatic.net/` by Michal Boleslav Měchura in 2011.

[6] *"MUSTE: Multimodal semantic text editing"* by Ljunglöf, Peter. `https://heatherleaf.github.io/muste/`

[7] `https://developers.google.com/blockly/`

[8] "Alice — Tell stories. Build games. Learn to program." `http://www.alice.org/`

a lot of the structure behind the CNL. The representation which the user works with ends up as some kind of tree-like structure combined with snippets of text embedded in it.

The idea of *frame-based editing* introduced in [96] also falls within this class, and aims to combine the advantages of block-based and text-based editing systems for programming languages. In essence, they propose a text-editing interface with additional visual markups to aid understanding, and template-and-menu editing of code to reduce syntax and type errors. The focus of this work is programming languages, and to our knowledge their approach has not been applied to CNLs.

5.5.2.3 Surface editing

By surface editing, we mean that the user is composing input phrases rather than constructing them from menus. This allows more freedom of input and the user may construct incorrect phrases which will later get rejected by the CNL parser.

The most basic kind of surface editor can be seen as a typical parser which allows arbitrary input but returns an error when it is not syntactically correct. This is familiar to us from the world of programming languages, and any CNL which has a parser written for it can provide this functionality. The quality and helpfulness of the supplied error messages can of course vary greatly, which directly impacts the user experience.

To improve on this, most CNL editors provide some kinds of cues to guide the user in the direction of writing something correct. The standard paradigm here is that of sequential left-to-right textual input with suggested completions for every word or phrase. The completions themselves may often be categorized and/or sorted in some way.

One such example is the WebALT project's WExEd tool for designing multilingual mathematics exercises, which uses TextMathEditor for input of individual phrases [37]. The input language here is a multilingual CNL for mathematics, which converts phrases into objects in the OpenMath formalism.

Attempto Controlled English (ACE), one of the best known modern CNLs, also has a few different editors for it which all follow this paradigm [95, 91, 85]. They provide predictive text editing where completions are split into 'function word', 'proper name', 'verb', 'variables', 'nouns', etc. along with some pop-up warnings when the user enters something incorrect. The editor has undergone a usability study [92], and the project itself has even produced a grammar notation for CNLs focusing on predictive editors and anaphora [93].

The GF runtime also provides incremental parsing, which has been used to create various predictive editors. The standard example of this interface is the Minibar[9], which has been used in various other applications (for example [30, 28, 31]).

[9] Minibar by Thomas Hallgren: `http://cloud.grammaticalframework.org/minibar/minibar.html`

Other CNL editors which also follow this paradigm include the ECOLE editor [151] for the PENG language, and its web-based successor for PENG*ASP* [67]. These editors include the ability to add out-of-vocabulary (OOV) words, support for anaphora across different phrases, and completions sorted by category in drop-down menus. The Ask Data Anything editor [152] uses a similar approach for allowing users to write complex database queries in a custom CNL. The editor also includes some automatic correction of erroneous input using fuzzy matching, rather than strictly preventing incorrect user input.

The KANTOO controlled language checker [113] is an editor for the KANT Controlled English (KCE) which takes a *prescriptive* approach: rather than forcing user input to be correct with respect to the CNL, the user is allowed free text input and is then given diagnostic information suggesting what can be improved, such as missing constituents, punctuation, or incorrect coordination between phrases.

This approach is also taken in the MuTUAL editor [115], designed for assisting non-professional writers in creating Japanese texts that conform to a set of writing rules for enabling translation to English. The tool allows free text input, detecting problems in the source text in real-time and providing diagnostic messages for interactive rewriting. The editor uses highlighting to indicate rule violations and prescribed terms, provides suggestions for alternate expressions and shows the CNL rules to aid users in making their text conformant. The tool also comes with an extensive user evaluation [114].

5.5.2.4 Search-based editing

In search-based editors the user inputs free text, but rather than being fed to a parser, this input is used to *search* for closely matching phrases within the CNL.

The work of [4] generates phrases from a CNL grammar, and combines this with a full text search engine. The advantage is that text-based search is well studied and one can use off-the-shelf search engines like Apache solar. However having to exhaustively generate phrases from a CNL can still be a bottleneck when the language is non-trivial in size, or even infinite as in our case.

In [143] it is presented a more advanced approach where instead of relying on exhaustive generation and a text search engine, phrases from both the input and the CNL are represented as vectors with infinite dimensionality. By considering only the non-zero elements, which are finite in number, closeness between input and valid CNL phrases can then be computed using cosine similarity. While this has been shown to work well on small CNL grammars, scalability is a problem because of the large number of comparisons which need to be made when searching for matches.

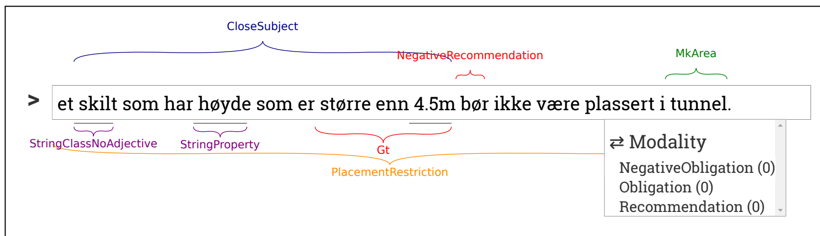


Figure 5.9: Example from the RailCNL phrase editor demonstrating the menu for substituting constructors.

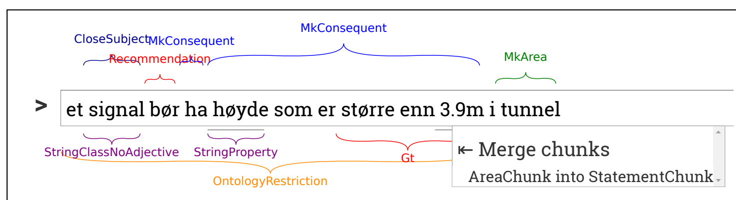


Figure 5.10: Example from the RailCNL phrase editor demonstrating the menu for merging chunks.

5.5.3 An editor for RailCNL

Taking clues from several of the approaches for building CNL editors described in Section 5.5.2, we developed an editor for Grammatical Framework languages with the specific use case of having railway regulations written by railway engineers using RailCNL. [10]. The figures in this section are actual screen-shots from the editor, and the input texts are therefore not translated into English.

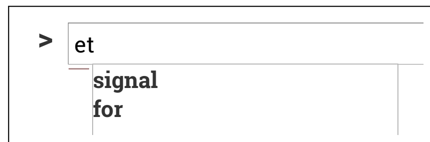
The RailCNL editor consists of a text input field containing the phrase which is being edited. There are no restrictions on the input to the text field, so the phrase can be empty, unparsable, partially parsable, or fully parsable. Whenever any change is made to the text, the parser will re-evaluate the text and update a drop-down menu and a partial parse tree visualization. The editor is thus mainly a surface editor (as described in Section 5.5.2.3), specifically an unrestricted text editor like mainstream programming language editors, but with two menu-based features: (1) a drop-down list giving menu choices relevant for the current text phrase and cursor position, and (2) a partial visualization of the parse tree where selected abstract syntax node types are drawn with a given a colour above or below the text input field. The drop-down menu may be compared to the auto-complete

[10]RailCNL editor prototype demo: `https://luteberget.github.io/ControlText/`

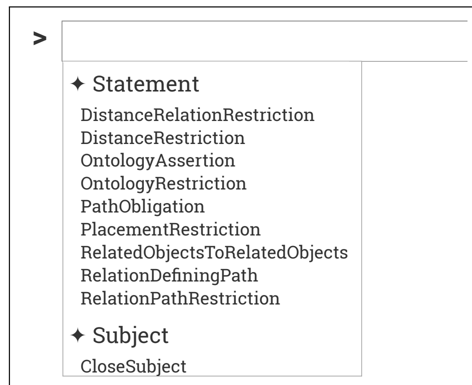
feature of mainstream programming editors, while the partial parse tree visualization can be seen as a hybrid between a full parse tree visualization and the kind of syntax highlighting used in mainstream programming editors.

The interaction between the text input, drop-down menu, and tree visualization works as follows:

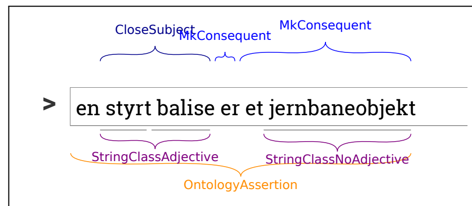
- Menu choices exist for any **concrete words** which are grammatically correct to insert at the current position. This works similarly to the auto-complete feature of mainstream programming editors, and the choices are supplied by the *predictive parsing* capability of GF.



- If the parsing has failed, and there is no top-level constructor of the parse tree, the menu will contain suggestions to **insert constructor**. When the text is empty, this part of the menu will suggest all top-level constructors, giving an overview of possible sentence structures. If the user chooses to insert constructors for which not all arguments are available in the current set of chunks, the smallest tree that has the required type is then constructed.



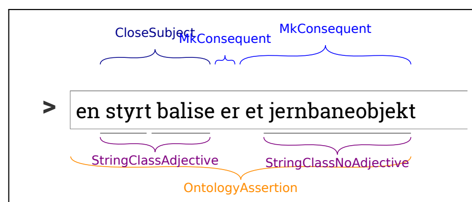
- When the parser has provided a full or partial parse tree, then a **partial parse tree visualization** is performed by consulting a list of selected types in the parse tree which are drawn as curly braces over the relevant part of the text using the *bracketed linearization* capability of GF. This allows the users to see while they are typing, what the parser is able to recognize.



- When the cursor is positioned inside a part of the text that has a valid parse tree, the menu shows suggestions for **substituting constructors**. All constructors which match the type of the tree nodes that cover the current cursor position have a corresponding list of alternative constructors, ranked by the number of terms that do not match in number and type. In the example below, the cursor is positioned on the word *"bør"* ("should" in English translation), which is tagged with the type *Modality*. Other modalities are suggested, giving an overview of possible expression types and thereby letting the user learn about the language as they are editing. See Figure 5.9.

- If more than one chunk is recognized, clicking the top-level constructor in one chunk will search for nodes in the trees of the other chunks where a constructor substitution would allow the current chunk to be inserted, thereby **merging chunks**. In the example below, the user has a valid phrase of type OntologyRestriction, and has tried to add a condition to the end saying that the restriction applies only in tunnels. However, the grammar specifies that the area modifier applies to the subject part of the sentence, and must therefore be moved further ahead in the sentence to be valid. When the cursor is placed on the area chunk, the editor will add a menu choice that will merge the area chunk into the statement chunk, producing a fully parsable phrase. See Figure 5.10.

- Dynamic vocabulary is used in the RailCNL system, for example, the user might define and name a new class, property or relation which is not part of the standard vocabulary. The grammar accepts this by allowing arbitrary strings in certain positions. Such unparsed words are highlighted in the editor by underlining. In the example below, "styrt balise" and "jernbaneobjekt" are part of the dynamic vocabulary.



The RailCNL editor works similarly to a programming source code editor. The design is based on the assumption that the user is a professional willing to learn about how the formal language and the verification system works. Even so, the editor's parse tree visualization and contextual menu-based operation encourages experimentation, exploration, and learning from examples.

The prototype RailCNL editor was implemented in JavaScript and HTML, which is suitable for demonstrations and entry into on-line databases, but a reimplementation using desktop application technologies should be considered to allow working with files on each engineer's local file system. The editor implementation uses the the grammar file produced by the Grammatical Framework for parsing and related functions through the Grammatical Framework runtime API. It also uses an editor-specific configuration file containing types and colours for the partial tree visualization, and for on-line documentation of constructors, which customizes the editor to the specific grammar.

5.6 Conclusions

RailCNL is our approach to *participatory verification*, where the end users (railway engineers, in our case) get full access to the verification properties. This allows them to actively participate in the verification by maintaining the rule base and managing their own properties (often based on experience and best practice). RailCNL formalizes, in a human-readable manner, relevant parts of the technical regulations and expert knowledge used in an on-the-fly verification engine integrated within railway construction design software.

We have collaborated with railway engineers associated with RailCOMPLETE during the design of the language and the writing of the verification properties. Their feedback on limitations in the coverage of the language and suggestions for simplification will continue to drive the design forwards.

We surveyed the Norwegian railway regulations and counted how much of the relevant regulations our basic RailCNL covers (see Section 5.4). The survey is limited to parts of the regulations covering railway track and signalling, as these are the disciplines that the RailCOMPLETE software development is currently focusing on.

RailCNL is implemented using the Grammatical Framework and its resource grammar library. While we have used Norwegian for representing regulations, RailCNL could be easily extended with other languages supported by the RGL. This would allow the system to be used for other authorities' regulations written in other languages. As long as most of the abstract syntax is re-used, the translation into Datalog should also be readily adaptable. The CNL literature, and Grammatical Framework specifically, contains a lot of explicit and implicit knowledge about constructing languages with natural syntax, and we have made explicit some of

this knowledge gathered from the CNL and GF community by describing details of the language design methodology in a general way.

A formal CNL with well-chosen linearizations can be very natural, and often perfectly readable for a non-programmer with the required domain knowledge; and we used this in Section 5.5.1 in our application to traceability of verification error messages. However, writing in a formal CNL can potentially be as difficult as writing in a programming language. A solution to this problem is the use of special-purpose editors which guide the user towards structuring their text according to the underlying formal grammar. Different approaches to CNL editors have been explored, which we reviewed in Section 5.5.2. We have been guided by these existing experiences when creating one such editor for RailCNL, presented in Section 5.5.3, which we plan to integrate in the RailCOMPLETE CAD environment, and carry out a usability study on its efficacy.

Focusing on *empathy* towards the intended user in the participatory verification process has shown us that integrated systems with familiar graphical user interfaces is an important requirement to engage non-programmer engineer users to adopt new tools into their daily routines. We hope that prioritizing according to the principles of participatory verification will in the long run help verification tools and techniques based on formal methods take hold in industrial practice, also outside the fields of software and electrical engineering.

5.6.1 Related work

Johannisson [82] describes a CNL targeting the Object Constraint Language (OCL) for use in reasoning about Java program correctness in the KeY system [11]. The language features dynamic vocabulary based on input UML diagrams where vocabulary updates are achieved by re-compiling the grammar using the GF compiler when needed. Angelov et al. [5] present a conflict detection framework where GF is used to map the contract language $\mathcal{CL}$ [137] into a CNL. Statement modalities, such as obligation, permission and prohibition, are applied to complex actions. The structure of the CNL is modelled after the $\mathcal{CL}$ language. Camilleri et al. [31] take a CNL approach to manipulating contract-oriented diagrams using a visual diagram editor, a CNL with text editor support, and a spreadsheet representation as interfaces to a common model, which can be translated into timed automata for reasoning about system properties.

Other efforts to define domain specific languages for railway verification have typically focused on the implementation of control systems, such as Vu et al. [168], while also considering the verification to be an activity which is separate from design and implementation. James et al. [81] show how to integrate UML modelling of the railway domain with graphical modelling and specification and verification languages, also keeping the focus on verifying the control system implementation of a fixed design.

5.6.2 Future work

In working with railway engineers, we discovered language features which could be added to increase the coverage of RailCNL:

1. A notion of scopes and exceptions, so that more complex conditional restrictions can be expressed more naturally.

2. Mathematical formulas as a sub-language.

3. Vague or soft requirements represented not for direct use in verification, but for requiring manual checks at some points.

We are continuing our collaboration with Norwegian railway engineers to evaluate the usability of our prototype tools, increase the text coverage and extend the language to handle other railway engineering disciplines such as catenary lines and ground works.

Some of the constructs in the CNL are highly specific to the text we are modelling, which is expected since the text freely uses a wide range of background railway knowledge, general engineering and mathematical knowledge. The main challenge in designing such a CNL is to find the underlying concepts, and to strike a balance between matching the level of abstraction on which the original text is based and introducing many special-purpose language constructs. More generally, any domain-specific language (DSL) must to some extent evolve alongside the needs of the applications it supports. See [58] for a general treatment of DSLs.

We envisage that RailCNL will evolve over time to include both new terminology appearing in new regulations as well as new knowledge and engineering practices. Therefore, the language would be maintained by engineers, maybe a proprietary version of RailCNL would be used internally by a company, including specific proprietary knowledge of the domain and practice.

5.7 Excerpts from RailCNL Grammar as written in GF

Complete source code at `https://github.com/luteberget/RailCNL.git`.

```
-- Overall grammar, combining modules.
abstract RailCNL = Statement,
 Ontology, Graph, Area ** {}
```

```
-- Grammar for expressions about
-- railway infrastructure layout.
abstract Layout = Ontology ** {
 cat DirectionalObject; GoalObject;
     PathCondition; SearchSubject;
 fun
  -- Convert subjects from Ontology
  -- module into search goals.
  AnySearchSubject : Subject
   -> SearchSubject;
  SameDirObject, OppositeDirObject,
  AnyDirObject : SearchSubject
   -> DirectionalObject;
  -- Specify restrictions on distance
  -- between sets of objects.
  DistanceRestriction : Modality
   -> Subject -> GoalObject
   -> Restriction -> Statement;   }
```

```
abstract Area = Graph ** {
  cat BaseArea; NamedArea; SingleArea;
    AreaConj; Area;
 fun
  -- Arbitrary area type from string.
  MkNamedArea : String -> Area;
(...)
  -- Use area as a Subject modifier.
  SubjectArea : OpenSubject -> Area
   -> Subject;
  -- Statement about area containment.
  PlacementRestriction : Modality
   -> Subject -> AreaConj
   -> Statement;  }
```

```
-- Partial concrete grammar in Norwegian
-- for the Ontology module.
concrete OntologyNor of Ontology = open
 SyntaxNor, ParadigmsNor,
 (RailLex = RailCNLLexiconNor) in {
 lincat (...)
   Class = CN; Property = CN;
   Subject = CN; Statement = Utt;
   Modality = {vv:VV; typ:ModalityType};
(...)
 lin
(...)
  -- Modalities
  Obligation
   = {vv = RailLex.must_VV; typ = MPos};
  NegativeObligation
   = {vv = RailLex.shall_VV; typ = MNeg};
  -- Apply restriction to ontology.
  OntologyRestriction mod subj cond =
   mkUtt (mkS
     (case mod.typ of {
       MNeg => negativePol;
       MPos => positivePol })
     (mkCl (forall_CN subj)
           (mkVP mod.vv cond)));   }
```

```
abstract Ontology = Statement ** {
 -- Partial grammar in the Railway
 -- CNL for expressing classes and
 -- properties of classes.
cat BaseClass; Class; Property;
 Value; ConsequentCondition;
 OpenSubject; Subject; Condition;
 Restriction; ClassRestriction;
 Modality; PropertyRestriction;
fun
 -- Class name from string.
 StringClass
  : String -> BaseClass;
 -- Class prefix string.
 StringClassAdjective
  : String -> BaseClass -> Class;
 -- Class name without prefix.
 StringClassNoAdjective
  : BaseClass -> Class;
 -- Property name from string.
 StringProperty
  : String -> Property;
 Gt, Gte, Lt, Lte, Eq, Neq
  : Value -> Restriction;
 -- Combine restrictions by 'and'/'or'
 AndRestr, OrRestr
  : Restriction -> Restriction
    -> Restriction;
 -- Combine property restrictions
 AndPropRestr, OrPropRestr
  : PropertyRestriction
    -> PropertyRestriction
    -> PropertyRestriction;
 -- Subject from Class and Condition
 SubjectCondition
  : Class -> Condition -> OpenSubject;
 -- Use class/property as condition
 ConditionClassAndPropertyRestriction
  : Class -> PropertyRestriction
    -> Condition;
 ConditionRelationRestriction
  : RelationMultiplicity -> Class
    -> Condition;

 -- Modalities: must/should and neg'd
 Obligation, NegativeObligation,
 Recommendation, NegativeRecommendation
  : Modality;

 -- Assertion statement about ontology
 OntologyAssertion
  : Subject -> ConsequentCondition
    -> Statement;

 -- Restriction statement
 OntologyRestriction
  : Modality -> Subject
    -> ConsequentCondition -> Statement;
}
```

Figure 5.11: Excerpts from RailCNL implementation in Grammatical Framework.

5.8 Example content from regulations

The following table lists some example excerpts from the regulations along with a translation into English, and a comment about use cases and relevance.

Original text	English translation	Comments
Source: **Overbygning**: 530 Prosjektering, Kap. 8 Helsveist spor, 2.1.		
De store krefter som kan forekomme i et helsveist spor stiller strenge krav til sporets konstruksjon.	The large forces that may occur in a welded track makes stringent demands on the track construction.	This sentence is not normative, and is unlikely to have any use in automated verification.
Source: **Overbygning**: 530 Prosjektering, Kap. 8 Helsveist spor, 2.1.3 a)		
Ballasten skal på linjen og i hovedspor på stasjoner være fullverdig grovpukk (av størrelse 31.5 – 63 mm)	The ballast on the line and in the main track at stations must be purely coarse crushed stone (size from 31.5 to 63 mm)	This is a specification which is absolute, and rules out the need for specifying this as a part of the design, because it is not part of a specific station. It can still be valuable to support this sentence in a CNL, and in a formal representation.
Source: **Overbygning**: 530 Prosjektering, Kap. 8 Helsveist spor, 2.1.2 a)		
Minste kurveradius for helsveist med betongsviller skal være 250 m.	The lowest allowable radius of curvature for whole welded track on concrete sleepers is 250 m.	This is a typical example of static infrastructure verification, expressible in Datalog as: error(Segment) :- trackSegment(Segment), trackSegmentRadius(Segment, Radius), Radius < 250.
Source: **Signal**: 550 Prosjektering, Kap. 6 Lyssignal, 2.1.2 j)		
Et innkjørhovedsignal skal plasseres $\geq$ 200 meter foran innkjørtogveiens første sentralstilte, motrettede sporveksel, se Figur 5.	A home main signal shall be placed at least 200 m in front of the first controlled, facing switch in the entry train path (see Figure 5).	This is the example that we have been using most frequently for the RailCons verification tool. Datalog: error(Sig,Sw) :- firstFacing(Bdry, Sw, Dir), homeSignalBetween(Sig, Bdry, Sw), distance(Sig, Sw, Dir, L), L < 200.

Original text	English translation	Comments
Source: **Signal**: 550 Signal, Kap. 5 Forriglingsutrustning, 2.8.1 Dekningsgivende objekt		
Følgende objekt kan være dekningsgivende: Hovedsignal, Dvergsignal, Sporveksel, Sporsperre, Avsporingstunge, Signal E35 Stoppskilt. Et hovedsignal skal vise signal ”Stopp” for å være dekningsgivende.	The following objects can provide flank protection: main signal, shunting signal, switch, derailer, derailing tongue, signal E35 stop sign. A main signal must display ”stop” to provide flank protection.	This regulation is relevant both for **specifying** the control system, and for verifying the **implementation**. The specification chooses which objects to use for flank protection (**static**) and what state they can be used in, while the implementation must correctly enforce the conditions saying which message the signal displays (**dynamic**).
Source: **Signal**: 550 Prosjektering, Kap. 6 Lyssignal, 2.1.2 i)		
Et hovedsignal bør ikke plasseres i tunneler, på bruer, eller andre steder hvor en eventuell togstans og dermed muligheten for avstigning, vil medføre fare.	A main signal should not be placed in tunnels, on bridges, or other places where halting trains and thus the possibility of disembarking, can impose dangers.	Here we have an example of a “*should*” modality, where the static infrastructure verification could issue a warning, but not an error. Also, it could be required to document the alternatives that were considered when deciding on the design.
Source: **Signal**: 550 Prosjektering, Kap. 5 Forriglingsutrustning, 4.1.1.1 i)		
For at en togvei skal kunne fastlegges, skal et objekt som gir dekning til togveien være dekningsgivende.	For a train route to be deactivated, any object giving flank protection must be in a protecting state.	This regulation concerns only the state of the control system, and as such relates to the implementation of the control system and not the static infrastructure specification.
Source: **Overbygning**: 530 Prosjektering, Kap. 5 Sporets trasé, 3.1 Dimensjonerende parametre		

Original text	English translation	Comments
See table below.	(a) minimum radius, (b) maximal superelevation, (c) limit on superelevation cause by derailment risk at low speeds, (d) limit for superelevation rate of change, (e) limit for superelevation deficit.	Limiting values are organized in a table for use in formulas in other sections.

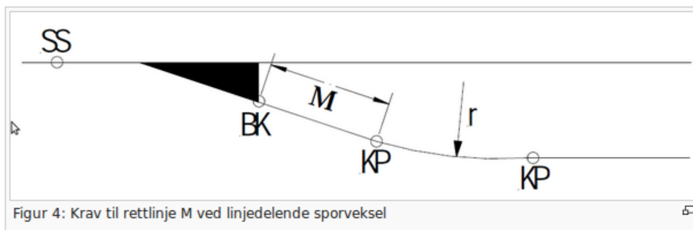
Tabell 2: Dimensjonerende parametre for nye baner og linjeomlegginger

	Symbol	Definisjon	Normale krav	Minste krav
a)	$R_{\min}$	minste radius	250 m	190 m
b)	h_{maks}	maksimal verdi for overhøyden [1)]	150 mm	
c)	h_{avsp}	grense for overhøyde pga. avsporingsfaren ved lave hastigheter	$\frac{R-100}{2}$ mm	
d)	$\left(\frac{\Delta h}{L}\right)_{maks}$	grenseverdi for rampestigning	1:400	
e)	I_{maks}	grenseverdi for manglende overhøyde [2)]	R ≤ 600: 100 mm R > 600: 130 mm	

Source: **Overbygning**: 530 Prosjektering, Kap. 5 Sporets trasé, 3.7 Sporveksler og sporforbindelser

Avstanden mellom sporveksel og overgangskurve, sirkelkurve, bru eller annen motstående sporveksel skal ikke være mindre enn avstanden M gitt i *Kurver uten overgangskurver*, krav b). M skal imidlertid ikke være kortere enn 6 m.	The distance between the switch point and the transition curve, circle curve, bridge or other opposite switch point should not be less than the distance M given in section *“curves without transition curves”*, requirements b). M shall not be shorter than 6 m.	The parameter M is explained by the figure below. Reference is given to another section of the regulations.

Original text	English translation	Comments



Figur 4: Krav til rettlinje M ved linjedelende sporveksel

Source: **Overbygning**: 530 Prosjektering, Kap. 5 Sporets trasé, 5 Største hastighet – sporets geometri		
Hastigheten i en kurve skal ikke være større enn: $V = 0,291 \cdot \sqrt{R\,(h + I_{\text{maks}})}$ (5) Hvis ligning 5 i tilfeller med falsk overhøyde gir lavere verdi enn 20 km/h gjelder $V = 20$ km/h.	The speed in a curve shall not exceed: $V = 0,291 \cdot \sqrt{R\,(h + I_{\text{maks}})}$ (5) If Eq. 5 gives a lower value than 20 km/h in situations with false superelevation, then $V = 20$ km/h shall be used.	Use of equations with designed and given parameters.

Source: **Signal**: 552 Vedlikehold, Kap. 6 Lyssignal, 3 Lyssignaler		
Dersom lyssignal er vridd eller på annen måte kommet ut av stilling skal dette utbedres snarest.	If a signal is twisted or in other ways are out of position, this shall be fixed as soon as possible.	Typical maintenance regulation. Here, it might be sufficient to identify this as a *checklist item*, for maintenance scheduling and reporting purposes.

5.9 Overview of Norwegian regulation contents

The technical regulations (*"Teknisk regelverk"*) can be found at `https://trv.banenor.no/` and consists of the following books:

- *Common regulations*: 501 Common regulations
- *Common electrical*: 510 Design and construction
- *Signs*: 515 Placement of signs along the track
- *Superstructure (tracks)*: 530 Design, 531 Construction, 532 Maintenance

- *Substructure*: 520 Design and construction, 522 Maintenance
- *Tunnels*: 521 Design and construction, 523 Maintenance
- *Bridges*: 525 Design and construction, 527 Maintenance
- *Overhead line*: 540 Design, 541 Construction, 542 Maintenance
- *Low voltage and 22 kV*: 543 Design, 544 Construction, 545 Maintenance
- *Power supply*: 546 Design, 547 Construction, 548 Maintenance
- *Signalling*: 550 Design, 551 Construction, 552 Maintenance, 553 Assessment
- *Telecommunications*: 560 Design and construction, 562 Maintenance

Structure of each book:

- Each book repeats the *common regulations* as the first three chapters.
- Following this will typically be a *general* section containing:
  - declaration of the scope of the book,
  - references to relevant standards,
  - definitions of relevant technical terms,
  - qualitative classifications, such as quality classes, risk classes, etc.
- The main part of a book consists typically of 5 to 10 chapters, each detailing a specific technical topic within the discipline. The text consists of:
  - Scope declarations
  - Definitions
  - Non-normative statements
  - Comments
  - Regulations (including tables and figures), with exceptions
  - Examples

The technical regulations contain a lot of generalities which are not necessarily normative, nor directly useful in a design setting. Based on the prioritized use case list, the following parts of the regulations should be considered first in designing and testing the formalization procedure:

1. Superstructure design (track design / *Overbygning: 530 Prosjektering*), especially regulations and formulas regarding

  - track geometry: curvature, gradients, etc.
  - switches: types, maximum speeds, naming (numbering), etc.

2. Signalling design (*Signal: 550 Prosjektering*), especially

  - signal placement, functions, sighting distance
  - train detector placement, classification
  - switch motors requirements and control system components
  - automatic train protection system (ATC) placement and functions
  - interlocking (control system) routes, conflicts, detection sections, safety classes, flank protection, overlaps, etc.

Drawing schematic plans 6

Engineering schematics of railway track layouts are used for several purposes: serving as construction blueprints, visualizations on train dispatch workstations, infrastructure models in timetabling software, specifications for interlocking control systems, and more. Because of the large distances involved, geographically accurate drawings are not always suitable for communicating an overview that can help with analyzing and reasoning about the railway models. Instead, many disciplines use *schematic* representations of the infrastructure, which provides a compressed overview, e.g., shortening sections of the railway that have low information density. Fig. 6.1 compares a geographically correct drawing against two alternative schematic renderings (for two purposes) of the same model. Producing schematic drawings like these involves practical and aesthetic trade-offs between intended structure, simplicity, and geographical accuracy.

Perhaps the most well-known railway schematics are the metro maps for passengers, popularized by the iconic Tube Map of the London Underground. When designing metro maps, removing and compressing geographical information better conveys topological structure (e.g., useful for finding transfers) and sequential information along lines (e.g., for finding your stop).

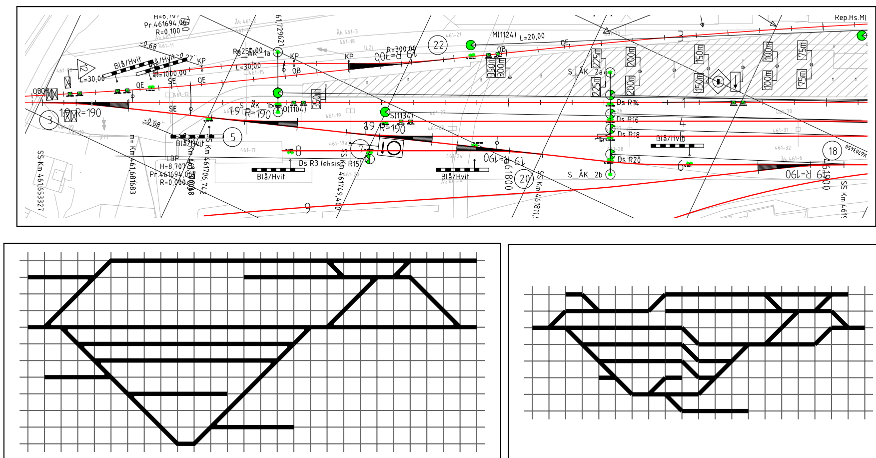


Figure 6.1: Example cut-out from a geographical railway drawing (top) and two corresponding schematic layouts, optimized for bends (bottom left) and optimized for height/width (bottom right). See on page 158 our tool's optimization options.

Methods for automatically producing metro maps have been surveyed in [174]. The main approaches are iterative and force-directed algorithms for gradually transforming a geographical network map into a simpler presentation [8, 27], and mixed integer programming methods for finding exactly grid-structured and rigidly optimized solutions [126, 128]. For railway drawings the convention is to use only horizontal, vertical, and diagonal lines (at $45°$). The problem of drawing graphs optimized for size and/or bends using only horizontal and vertical lines (so-called orthogonal drawings) can be solved by efficient algorithms [161], but adding diagonal lines in general makes the problem NP-complete [126, 127].

Schematic railway drawings used for engineering are usually more strictly constrained than metro maps, but still have large variety in different versions produced for different engineering use cases, project stages, and operational scenarios. Especially in construction projects for new railway lines or upgrades, frequent changes are made in coordinated 2D, 3D, geographical, and schematic models of the railway infrastructure, which can cause much repeated manual work in updating and cross-checking these models after every change in the construction design work in several engineering and construction categories, such as tracks, signaling and interlocking, catenary, cables, telephony.

Automatically producing consistent and high-quality schematics from other models has great potential to increase the efficiency and quality of the documentation, speed up cross-discipline communication during design and construction phases, and also opens up for easier data transfer to other tools.

In this chapter we develop methods for producing a type of schematic track plan which is suitable for infrastructure within a single corridor, meaning that each point on each track can be located on a common linear axis. We call this a *linear schematic drawing* (see Definition 1). This is a common drawing used for many purposes in construction projects, where drawings typically show placement of tracks and track-side equipment on a single station or along a single corridor. More generally, this problem concerns network structures that are oriented along a linear axis, such as highways, railways, or public transit systems, but may also be extended to encompass routing in electronic design (see e.g. the problem description for VLSI routing in [130]). On larger scales with multiple corridors, the visualization may be split into individual corridors, as in our setting, but for some applications, such as an overview of a national railway network or a city metro network, the single corridor assumption will not work well, and other approaches (see e.g. [126, 128]) may be more relevant.

Linear schematic drawings specifically have little coverage in the literature. A specialized algorithm presented in [23] computes corridor-style drawings, but does not guarantee that switch shapes are preserved, and does not offer choice in optimization criteria. For comparison, we apply our method to examples taken from [23] (see Fig. 6.11). Another algorithmic approach described in [153] has similar goals, but does not automatically produce high-quality results and relies on inter-

active guidance from the user and manual post-processing.

Graph drawing techniques (see [45, 46] for a general overview) have been developed for a great number of different use cases. Most closely related to engineering drawings are the orthogonal layout methods (see e.g. [133, 132, 161]). However, most approaches from the graph drawing literature, including orthogonal layout methods, produce outputs that have a distinct style and are not suited to be customized to adhere to engineering drawing conventions.

Instead, we have solved the problem by modeling engineering drawings as mathematical optimization problems using constraints formulated as Boolean satisfiability and difference constraints. We present how different available constraint programming systems can be used to express our constraints, solve optimization problems, and produce high-quality engineering drawings.

The main contributions of this paper are: (1) We describe and formalize the problem of *linear schematic railway drawings* in Section 6.1. (2) We define three mathematical models for schematic plans, and compare their strengths and weaknesses in Section 6.2. (3) We develop a downloadable tool that can be used by railway engineers to visualize infrastructure, and demonstrate its performance and output on real-world infrastructure models in Section 6.3. Our tool is meant to be used as a module integrated in the RailCOMPLETE engineering framework; but it can also be used as a standalone tool by researchers and developers working on new techniques for analysis and verification, e.g. on interlockings or capacity and timetabling, who can greatly benefit from low-effort, high-quality visualizations in order to improve communication, usability, and for lowering the barrier for adoption of their tools and techniques. Our tool takes input railML files, which are widely available among railway engineers as it is a standard description format for railway infrastructure. The tool also has options for placing symbols besides a track in the schematics.

6.1 Problem definition and formalization

6.1.1 Linear positioning system

It is a common practice in railway engineering to use a linear reference positioning system, which assigns a scalar value to each point on, or beside, a railway track. The value corresponds approximately to the traveling distance along a railway corridor from a reference point (which is often a known point on the central station of the network). For a single track, the linear reference system may simply be the arc length from the start of the track's center-line curve. Double tracks and side tracks are typically mapped to the linear reference position by geometrically projecting each point onto a reference curve. The projection's target curve may either be a selected *reference track* (see Fig. 6.2), or another curve that does not necessarily coincide with a track, such as the geometrical center-line of the corridor. For the

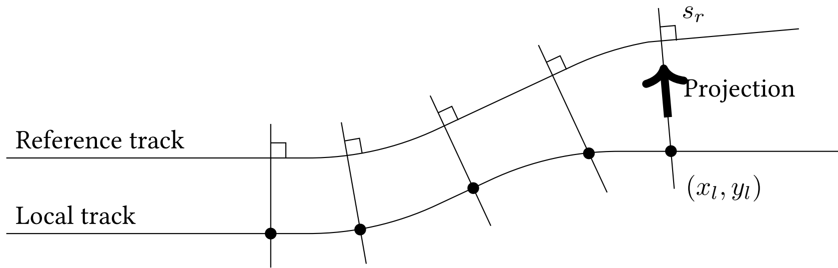


Figure 6.2: Linear reference position calculated by projection onto a reference track.

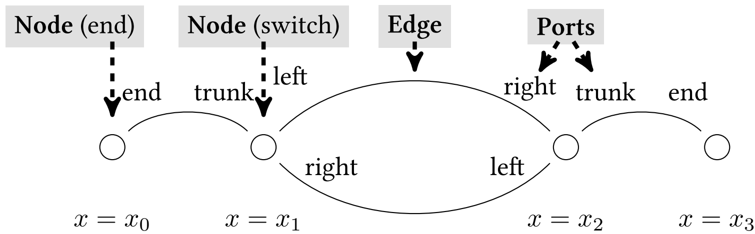


Figure 6.3: Graph representation of linearized track plan. Nodes are ordered by an x coordinate, and have a given type which determines which ports it has, e.g., a switch node has *trunk*, *left*, and *right* ports. Edges connect ports on distinct nodes.

rest of this paper, we assume that all locations are already given in such a linear reference system.

6.1.2 Track network representation

Different track segments are connected together at *switches* in a graph-like network. The mathematical definition of a graph is too abstract for many engineering use cases. Some applications use a *double node graph* [78], or describe tracks as nodes with two distinct sides [1]. For a schematic plan, we model switches and crossings as graph nodes which have a given set of *ports* (Fig. 6.3 presents all our modeling elements). Each end of each edge connects to a specific port on a specific node. Model boundaries and track ends are also represented as nodes with a single port.

Each location where tracks start/end or intersect with other tracks is represented as a node of a given class. The classes used in this paper are ends, switches, crossings, and flyovers (shown in Fig. 6.4 with all their representative variants). Each class comes with a different set of drawing requirements. For example, a

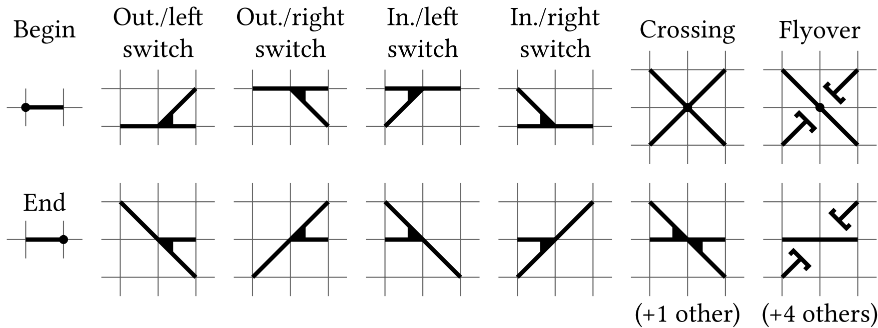


Figure 6.4: Node classes and their drawing variants. Begin/end nodes have one variant each. Switches are divided into four classes (each with two variants) based on their orientation (incoming or outgoing) and their course (deviating left or right). Crossings have three variants, and flyovers have six variants (symmetric variants omitted).

switch is oriented such that its branching edges (left/right) point either up (called an outgoing switch) or down (called an incoming switch), seen in the positive direction of the linear positioning system, and each switch class can be drawn in two different variants, chosen freely, one with the trunk and straight leg directed horizontally and another with the deviating leg directed horizontally.

6.1.3 Linear schematic drawing

A linear schematic drawing algorithm is a core concept in our formalization.

Definition 1. *A linear schematic track drawing algorithm* $d : (N, E) \to L$ *assigns a set of line segments* L *to each edge in the set* E *of edges connecting the set of nodes* N*, where:*

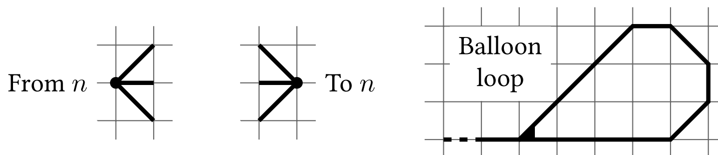
- $N = \{n_i = (c_i, s_i)\}$*, where* $c_i \in C$ *is a node class, and* $s_i \in \mathbb{R}$ *is a linear position distinct from other nodes' positions.*

- $E = \{e_j = (n_a, p_a, n_b, p_b)\}$*, where* $n_a, n_b \in N$ *are two nodes where* $s_a < s_b$ *and* p_a, p_b *are distinct, available ports on the referenced nodes.*

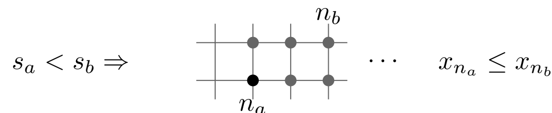
- $L = \{(e_j, l_j)\}$*, where* l_j *is a polyline, representing the drawing of edge* $e_j \in E$*, and defined by a sequence of points in* $\mathbb{R}^2$*,* $\langle (x_1^j, y_1^j), (x_2^j, y_2^j), \ldots, (x_n^j, y_n^j) \rangle$*. The polyline consists of the line segments connecting consecutive points in this sequence.*

Of course, the definition of a track drawing algorithm in itself does not ensure that the output drawing is suitable for reading. To ensure a usable output we establish a set of *criteria for drawing quality* based on engineering conventions and aesthetic judgments. We divide the criteria into *hard constraints*, that all drawings must satisfy and that we can base mathematical models on, and *soft constraints*, which are optimization criteria that can be prioritized differently in different use cases. We base our models on the following hard constraints provided by railway signaling engineers (from Railcomplete AS):

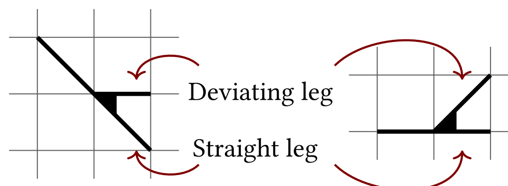
(A) **Octilinearity**: the lines representing tracks should be either horizontal, or diagonal at 45°. This property contributes to the neat look of a schematic drawing, and also gives a visual clue that the drawing is not fully geometrically accurate. If loops are present in the infrastructure, vertical lines may also be allowed, such as in the *balloon loop* used on many tram lines.



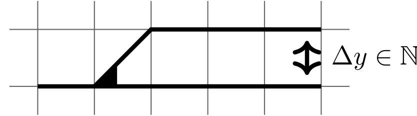
(B) **Linear order**: the reference mileages of locations on the infrastructure should be ordered left-to-right on the schematic drawing to give a clear sense of sequence, which is useful when navigating the infrastructure and reasoning about train movements.



(C) **Node shapes**: switches split the track on the *trunk* side into a left and a right leg on the *branch* side. Left and right should be preserved so that the layout can be traced back to the geography. Also, one of the legs of the switch is typically straight and the other is curved, so typically it is also desirable to preserve the straight leg's direction relative to the trunk.



(D) **Uniform horizontal spacing**: parallel tracks are typically required to be drawn at a specific distance from each other, which we normalize and say that y coordinates take integer values. Note that x coordinates have no such restriction, but consecutive nodes will often be placed at integer-valued distances to fulfill the octilinearity constraint.



Even with the above constraints fulfilled, there is no guarantee that the drawing output of an algorithm can be deemed of high-quality. For this we use the following soft constraints as optimization criteria:

(i) **Width** and **height** of the drawing.

(ii) **Diagonal line length**, the sum of length of non-horizontal line segments.

(iii) **Number of bends**, i.e. the number of direction changes on lines.

These criteria have different priorities in different use cases. For example, a signaling schematic might be optimized to have a minimum amount of diagonal lines to neatly show several concurrent train movements and their relative progress, while a dispatch control station schematic might be optimized for width to fit more infrastructure into a computer screen.

Several or all of the criteria can be combined into an optimization objective, either by a scoring function, or more commonly, by simply ordering the objectives and performing lexicographical optimization on each objective in turn. Our tool (detailed in Section 6.3) provides options for ranking the objectives.

6.2 Model definitions and drawing algorithms

This section describes three different models of linear schematic drawings. First, we present a linear programming formulation where edges can have up to two bends. The resulting optimization problem is efficiently solvable, but has some drawbacks in visual quality. Second, we introduce Boolean choice variables to mitigate the shortcomings of the linear programming formulation, and use instead a SAT solver and lazy solving of difference constraints to optimize the Boolean/numerical model (keeping the maximum of two bends per edge). Finally, we present a different Boolean model with unbounded number of bends per edge, which makes this formulation able to optimize drawing size further than the two previous models. However, this comes at the cost of increased running time. Comparison Figures 6.7, 6.8, and 6.11 demonstrate the strengths and weaknesses of each approach,

while Table 6.1 shows their relative performance. All models use a pre-processing step which orders edges vertically, as in Sec. 6.2.1.

6.2.1 Vertical ordering relation on edges

From the nodes and edges defined as inputs to the linear schematic drawing algorithm, it is possible to derive a vertical ordering relation $<_E$ on the set of edges. This relation is a strict partial order relating edges whose linear position intervals overlap, i.e., it relates each pair of edges e_a from n_{a_l} to n_{a_r}, and e_b from n_{b_l} to n_{b_r}, for which the real-valued segments are intersecting

$$]s_{a_l}, s_{a_r}[\ \cap \ ]s_{b_l}, s_{b_r}[\ \neq \emptyset.$$

Such a relation can be established by considering paths starting in each of the branch-side ports of each switch, crossing, and flyover (cf. Fig. 6.4). For example, an outgoing switch with branch-side edges e_a and e_b connecting to its right and left ports, respectively, will obviously have $e_a <_E e_b$. Each edge connected to the outgoing edges from the other side of e_a and e_b will also be ordered vertically, and so on until either of the following termination conditions are fulfilled:

(C1) The two sets of edges meet in another node.

(C2) One of the sides has no more edges to follow.

More precisely, we define $<_E$ by the following. Let $G = (N, E)$ be the graph from Definition 1. We first look in the positive direction on the linear reference axis. We define a vertical order relation $<^i_E$ for each node $n_i \in N$. If n_i has less than two ports on the side of increasing linear position, $<^i_E$ is empty. However, if the node has two ports on the side of increasing linear position, let the edges connected to these ports be e_l, the lower edge, and e_h, the higher edge. For example, in an outgoing switch node (cf. Fig. 6.4), these correspond to the right and left ports, respectively.

For any node n_j with $s_i < s_j$, define the directed graph $H_{]i,j[}$ containing:

- The subset of nodes from G with positions in the open interval $]s_i, s_j[$, along with any number of fresh nodes (i.e. the nodes n_i and n_j are not included).
- The subset of edges from G which have at least one end connected to a node from the open interval $]s_i, s_j[$, directed in the direction of increasing linear position. If an edge connects to a node from G which is not included in $H_{]i,j[}$, that connection is replaced with a connection to a distinct fresh node.

We are looking for those nodes n_j such that, in $H_{]i,j[}$, the set of reachable edges when starting from e_l are disjoint from the set of reachable edges when starting from e_h (termination condition (C1)), see Fig. 6.5(a). Also, the linear position interval of each edge reachable from e_l should have a non-empty intersection with at

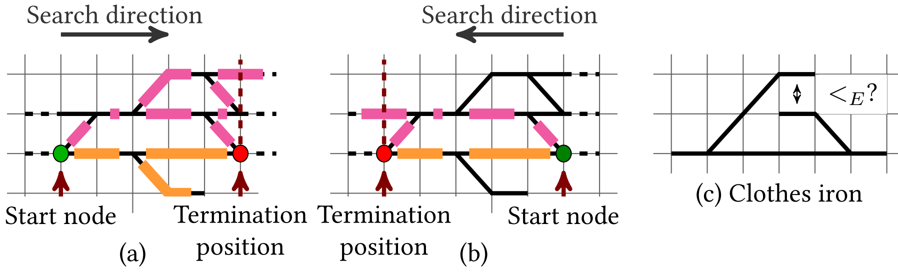


Figure 6.5: A search procedure starting in each node produces a set of tuples for the edge vertical order relation $<_E$. Figures (a) and (b) show two different start nodes and search directions, where the lighter, orange edges are all below darker, magenta edges. Figure (c) shows an input on which the procedure cannot decide an ordering.

least one edge reachable from e_h, and vice versa (termination condition (C2)). The node n_j which has the highest position s_j while still fulfilling the above criteria, is called the *termination position*.

Each edge e_x reachable from e_l in $H_{]i,j[}$ is below all edges e_y reachable from e_h in $H_{]i,j[}$ whenever this pair of edges has overlapping linear position intervals, in which case we have $e_x <_E^i e_y$.

For the direction of decreasing linear position we apply the same argument with horizontal directions reversed (see Fig. 6.5(b)). Finally, the relation $<_E$ is defined as the union of the relations from each node,

$$<_E \; = \bigcup_{n_i \in N} <_E^i .$$

Unconnected graph components must still be explicitly ordered, and the same for some connected topologies such as the clothes iron example in Fig. 6.5(c). These are usually easy to decide from, e.g., a geographical model, and this situation occurs rarely, in our experience.

6.2.2 Level-based linear programming encoding

We start out by giving a constraint system on linear equations over continuous numerical variables which fulfills the hard requirements from Section 6.1.3 and can be solved efficiently by linear programming. We used the CBC solver v2.9 [38]. Later, the shortcomings of this model will motivate the introduction of Boolean and integer-valued variables and a SAT problem formulation.

For each node n_i we use two real variables, x_i and y_i, representing the schematic coordinates of nodes. For each edge e_i we use one real variable l_i representing the

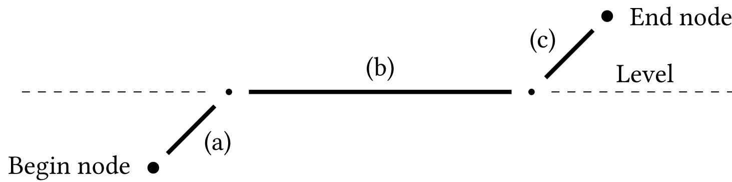


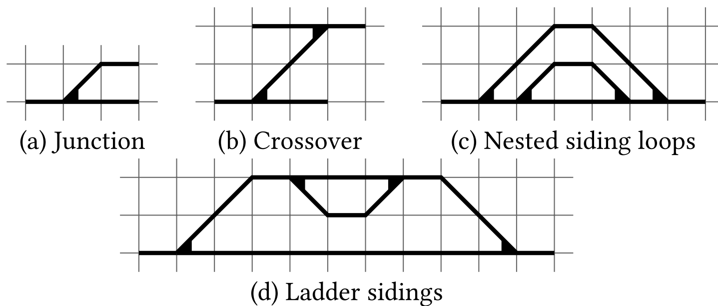
Figure 6.6: The *edge level model* divides the edge into three sections on the horizontal axis: (a) the initial diagonal section from the left-most node to the edge level, (b) the middle horizontal section connecting the two diagonal sections, (c) the final diagonal section reaching the right-most node from the edge level. Any of these may have zero length.

edge's *level*. This builds in an assumption that each edge is drawn in three parts as explained in Fig. 6.6. We introduce the following constraints:

1. Node location ordering for successive nodes n_i, n_j gives $x_i \leq x_j$, corresponding to the linear order requirement (from Sec. 6.1.3(B)).
2. Node location distance for nodes n_i, n_j connected by an edge e_k, where $s_i < s_j$, gives $x_i + |l_k - y_i| + |y_j - l_k| + q_k \leq x_j$, where q_k is 0 if the edge connects an outgoing switch to an incoming switch with the same branching direction, and 1 otherwise. This creates room for a horizontal line segment if needed. The sign of the absolute value terms is determined statically (not part of the linear programming) by the node class and variant. This constraint corresponds to the octilinearity requirement (from Sec. 6.1.3(A)).
3. Edge level ordering for edges: $e_i <_E e_j$ gives $l_i + 1 \leq l_j$, corresponding to the node shape requirement (from Sec. 6.1.3(C)).
4. Edge levels are related by switches, i.e.: each switch node n_i constrains the trunk-side edge e_j and the straight branch-side edge e_k to be at the same level as the node ($y_i = l_j = l_k$) corresponding to the node shape requirement.

Note that the uniform horizontal spacing constraint (from Sec. 6.1.3(D)) is implicit in these equations. Now we have the following criteria available for optimization:

- **Width of the drawing.** Take the node n_i with the lowest s_i, and the node n_j with the highest s_j. Then the width of the drawings is $x_j - x_i$.
- **Height of the drawing.** The height of the drawing is not directly expressible in this model, but can be approximated by summing the vertical level difference of edges. For pairs of edges e_i, e_j where $e_i <_E e_j$, the vertical level difference distance is $l_j - l_i$.



(a) Junction (b) Crossover (c) Nested siding loops

(d) Ladder sidings

Figure 6.7: Output examples for the linear programming method. The junction (a) and nested sidings (c) are correctly drawn. The crossover (b) has a 2 unit diagonal edge height, where 1 unit would be sufficient – this is caused by each edge having a vertical *level* distinct from its neighbors. The ladder sidings (d) are unnecessarily wide because the model does not handle node shape variants (compare with Fig. 6.8(b)).

Some output examples from the linear programming solution are shown in Fig. 6.7. Although efficiently solvable, this linear programming solution has a main drawback in that it is not able to choose between different alternatives for drawing a node. For example, in the so-called ladder configuration shown in Fig. 6.7(d), much space is wasted on diagonal lines going to the top-most level, when the two topmost switches could have been rotated to produce a simpler drawing. Also, each edge needs to have a y-value distinct from its neighbors, even if it is drawn only with diagonals, such as in Fig. 6.7(b), which contributes to inefficient use of space. Both these shortcomings will be improved by the level-based Boolean formulation in the next section.

6.2.3 Level-based SAT encoding

We reformulate the problem using variables from the Boolean and bounded integer domains. Since we are dealing with small integers, we can transform the problem into a Boolean satisfiability problem (SAT) by encoding numerical variables into Boolean variables and use incremental SAT solvers which can be efficient for lexicographical optimization on small discrete domains, as ours.

Integers can be encoded into SAT in various ways. Eager encodings represent numbers and constraints directly using a set of Boolean variables and constraints and creates an equisatisfiable SAT instance. Most commonly used is the binary encoding (one Boolean for each bit) and the unary encoding (one Boolean for each distinct number). See [15] for details. Lazy encodings, as used in SMT solvers (see [10, 123] for an introduction), can avoid some of the work of transforming and

solving a large SAT problem by abstracting the numerical constraints into marker Boolean variables. Only when the SAT solver sets markers to true, another procedure (the *theory solver*) will go to work on the numerical constraints and report unsatisfiable combinations back to the SAT solver.

Although the SAT problem itself does not directly concern numbers, much less numerical optimization, an *incremental* interface to a SAT solver allows solving many similar problems consecutively. For a set of constraints ϕ, we can perform numerical optimization on some number x by solving the sequence of formulas $\phi \wedge (x < m_1)$, $\phi \wedge (x < m_2)$, $\dots$, where the sequence m_i is a linear or binary search over the range of x, locating the smallest value that satisfies the constraints. Querying the solver successively with such similar formulas incrementally is much faster than solving the instances separately.

We used the MiniSAT [52] solver v2.2.0 with unary encoding of bounded integers and also lazy representation of unbounded integers with difference constraints, i.e. constraints of the form $x_i - x_j \leq k$, where k is a constant. Difference constraints are suitable as a first-line refinement in SMT solvers (see e.g. [22]) because they can be efficiently solved.

We keep the assumption from the previous subsection that each edge is assigned to a single level, and extend the problem representation as follows:

1. Distances between nodes are represented as a saturating unary number of size 2, i.e. $\Delta x \in \{0, 1, \geq 2\}$. This allows us to distinguish between short ($\Delta x \leq 1$) and long ($\Delta x \geq 2$) edges.
2. For each edge e_j, we use Booleans q_j^{up} and q_j^{down} to indicate a short edge pointing up/down, respectively, seen in the direction of increasing x.
3. Node vertical coordinates y_i and edge levels l_j are represented by unbounded integers on which we can conditionally impose difference constraints.
4. Variant selection $r_i \in R(c_j)$ for each node i indicates the node's variant from the available shapes $R(c_j)$ of the node class $c_j \in C$ listed in Fig. 6.4.
5. Edge direction values, $d_i^{\text{begin}}, d_i^{\text{end}} \in \{\text{Up}, \text{Straight}, \text{Down}\}$, for the beginning and end of each edge e_i, are based on node variant values.

We need the following constraints:

- Each edge must be at least 1 unit long on the x axis.
- Edge ordering constraints for $e_a <_E e_b$:

$$l_a \leq l_b, \quad \left(\neg q_a^{\text{up}} \wedge \neg q_b^{\text{down}}\right) \Rightarrow l_a + 1 \leq l_b$$

  If an edge is a short edge (such as a crossover between two adjacent tracks) it does not require its own level, and we use instead the same level as the one of its

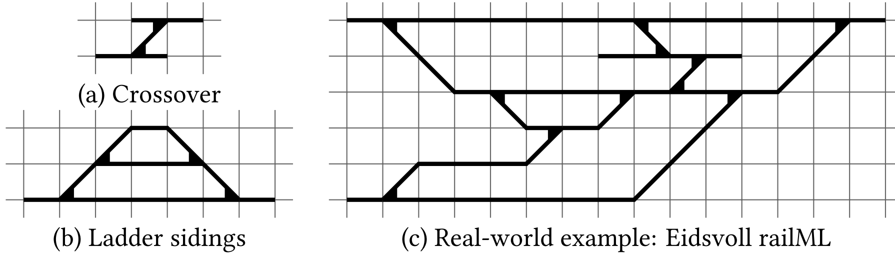


Figure 6.8: Output examples for the level-based SAT method. The crossover (a) requires only a 1 unit diagonal edge (improving Fig. 6.7(b)). The ladder sidings (b) now use diagonal switch variants to improve width, height, and bends (improving Fig. 6.7(d)). The Eidsvoll station (c) demonstrates real-world infrastructure imported from railML.

end nodes which has the *highest* value. This allows to produce a better crossover drawing, as in Fig. 6.8(a) instead of Fig. 6.7(b).

- An edge i is short (q^{up} or q^{down}) if both ends have the same direction and the vertical distance between nodes is one:

$$q_i^{\text{up}} \Rightarrow (d_i^{\text{begin}} = \text{Up}) \wedge (d_i^{\text{end}} = \text{Up}) \wedge (y_a + 1 = y_b)$$

$$q_i^{\text{down}} \Rightarrow (d_i^{\text{begin}} = \text{Down}) \wedge (d_i^{\text{end}} = \text{Down}) \wedge (y_a - 1 = y_b)$$

- Direction on edge i decides vertical level constraints:

$$(d_i^{\text{begin}} = \text{Straight}) \Rightarrow (y_a = l_i), \quad (d_i^{\text{begin}} = \text{Up}) \Rightarrow y_a + 1 \leq l_i,$$
$$(d_i^{\text{begin}} = \text{Down}) \Rightarrow \big((q_i^{\text{up}} \Rightarrow (y_a \geq l_i)) \wedge (\neg q_i^{\text{up}} \Rightarrow (y_a \geq l_i + 1))\big)$$

  And correspondingly for d^{end}.

- The sum of Δx values over the edge must match the shape of the edge:

$$\left(q^{\text{up}} \vee q^{\text{down}}\right) \Rightarrow \Sigma_{j \in (a,b)} \Delta x_j \leq 1$$

$$\left(\neg q^{\text{up}} \wedge \neg q^{\text{down}} \wedge \left(d^{\text{begin}} \neq \text{Straight} \vee d^{\text{end}} \neq \text{Straight}\right)\right) \Rightarrow \Sigma_{j \in (a,b)} \Delta x_j \geq 2$$

Since the shape of an edge is now explicit through d^{begin} and d^{end}, we can optimize for the number of bends to produce Fig. 6.8(b) instead of Fig. 6.7(d).

The level-based representations do not represent the shapes of edges explicitly at each coordinate, and thus cannot insert bends at arbitrary locations, something

which is needed to pack drawings together more tightly. A straight-forward grid-based SAT encoding could associate each point on a grid with a choice of any node, and each cell with a choice of edge shape. With this encoding, however, drawings with only about 30 nodes take hours to optimize. We do not describe this method in more detail here, but we have implemented it and tested its performance compared to the other methods, as shown in Table 6.1.

6.2.4 Cross-section SAT encoding

Instead of directly representing a grid, we define a vertical cross-section c_k of the drawing, represented by a unary-encoded integer $y^k_{e_i}$ capturing the height of each edge e_i at some horizontal location in the drawing. This naturally allows us to use the edge vertical order $<_E$ as constraints on unary numbers $y^k_{e_i} <_E y^k_{e_j}$. Each pair of successive nodes is transformed into a sequence of such cross-sections, and we associate a direction $d^k_{e_i} \in \{\text{Up}, \text{Straight}, \text{Down}\}$ with each edge e_i at each cross-section c_k, giving the shape of the edge to the *left* (lower x value) of the cross-section. Cross-sections can be enabled or disabled (represented by b_k) to optimize the width of the drawing. Finally, the *ahead* Boolean $a^k_{e_i}$ for each edge at each cross-section marks whether the shape of the edge has already been constrained for the next cross-section to the right (higher x value), which allows nodes to impose edge shape constraints in both x-axis directions.

With this representation, we can impose constraints as follows:

1. Edge vertical order:

$$(e_i <_E e_j) \Rightarrow \bigwedge_{c_k} y^k_{e_i} \leq y^k_{e_j}$$

2. A begin node at cross-section c_k constrains the edge shape to the right, and makes the y value unequal to the y value of other edges $e_j \in c_k$.

$$a^k_{e_i} \wedge d^k_{e_i} = \text{Straight}, \qquad \bigwedge_{e_j \in c_k} y^k_{e_i} \neq y^k_{e_j},$$

   and similar for end nodes, in the opposite direction:

$$\neg a^k_{e_i} \wedge d^k_{e_i} = \text{Straight}, \qquad \bigwedge_{e_j \in c_k} y^k_{e_i} \neq y^k_{e_j}.$$

3. A switch node at cross-section c_k constrains the edge shape in both directions by constraining the incoming edges e_i according to the node class variant. For example, for an outgoing left switch we have one incoming edge e_{i1}:

$$\neg a^k_{e_i1} \wedge d^k_{e_i1} \neq \text{Up}$$

The incoming edges e_i are replaced by the outgoing edges e_j in the cross-section representation. For example, for an outgoing left switch (see Fig. 6.9) we have two outgoing edges e_{j1}, e_{j2} as the left and right ports, respectively:

$$a^k_{e_{j1}} \wedge a^k_{e_{j2}},$$

and we have two choices of shape:

$$\left(d^k_{e_i} = \text{Straight}\right) \Rightarrow \left(y^k_{e_i} = y^k_{e_{j2}} \wedge d^k_{e_{j2}} = \text{Straight} \wedge d^k_{e_{j1}} = \text{Left}\right)$$

$$\left(d^k_{e_i} = \text{Down}\right) \Rightarrow \left(y^k_{e_i} = y^k_{e_{j1}} \wedge d^k_{e_{j2}} = \text{Down} \wedge d^k_{e_{j1}} = \text{Straight}\right)$$

Constraints are similar for other node classes.

4. Disabled cross-sections propagate all their values:

$$\neg b_k \Rightarrow \bigwedge_{e_i \in c_k} \{y^k_{e_i} = y^{k+1}_{e_i} \wedge a^k_{e_i} = a^{k+1}_{e_i} \wedge d^k_{e_i} = d^{k+1}_{e_i}\}$$

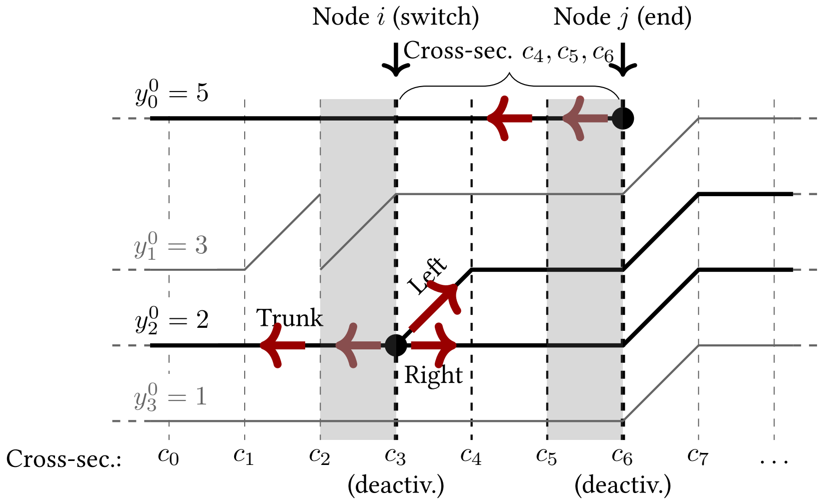


Figure 6.9: Cross-section SAT representation. Dashed vertical lines show cross-sections c_i. Edges have a y value and a direction to the left of each cross-sec. Thick red arrows are constraints imposed by node type. Gray columns correspond to deactivated cross-sections, where shape constraints are propagated to the next or previous column.

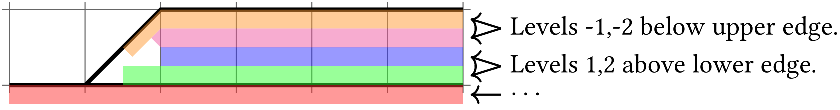


Figure 6.10: Label placement can be done by restricting symbols to fit into a set of levels above and below each line, which reduces the constraints to linear ordering.

5. Enabled cross-sections require consistency between edge shapes and y values:

$$b_k \Rightarrow \bigwedge_{e_i \in c_k} \left\{ \left(\neg a_{e_i}^k \wedge d_{e_i}^{k+1} = \text{Up} \right) \Rightarrow y_{e_i}^k + 1 = y_{e_i}^{k+1} \right\}$$

   And correspondingly for *Straight* and *Down* directions.

6. Enabled cross-sections realize rightward-constrained *ahead* values a:

$$b_k \Rightarrow \bigwedge_{e_i \in c_k} \left\{ \left(a_{e_i}^k \Rightarrow y_{e_i}^k = y_{e_i}^{k+1} \right) \wedge \left(a_{e_i}^k \Rightarrow d_{e_i}^k = d_{e_i}^{k+1} \right) \wedge \neg a_{e_i}^{k+1} \right\}$$

With this formulation we can choose freely between prioritizing width, height, or bends, and the resulting plans have lower total width than for the level-based methods, since the grid-based method has the added freedom of inserting bends at any location along an edge. See Fig. 6.11 for a comparison.

6.2.5 Symbols and labels

A railway engineering schematic often features a large amount of different symbols and labels (see the example in Fig. 6.1). In some cases, the symbols and labels can be placed onto a well laid-out track plan without needing to change the track plan, but there are common cases where the track layout must be drawn in a way that accounts for the amounts and sizes of symbols and labels. Our tool has options for placing symbols into two rows above and below each track, which is suitable for signaling drawings.

Label placement in general is known as a hard problem in graph visualization. We use a simplified approach suitable for thin rectangular symbols (e.g. as in Fig. 6.1), and assign each symbol to a level above or below the track (see Fig. 6.10). Difference constraints on x values ensure that symbols are ordered and not overlapping. When constraints are satisfied, we use linear programming to minimize the deviation from proportional distance between nodes, so that symbols are close to each other on the drawing if they are physically close.

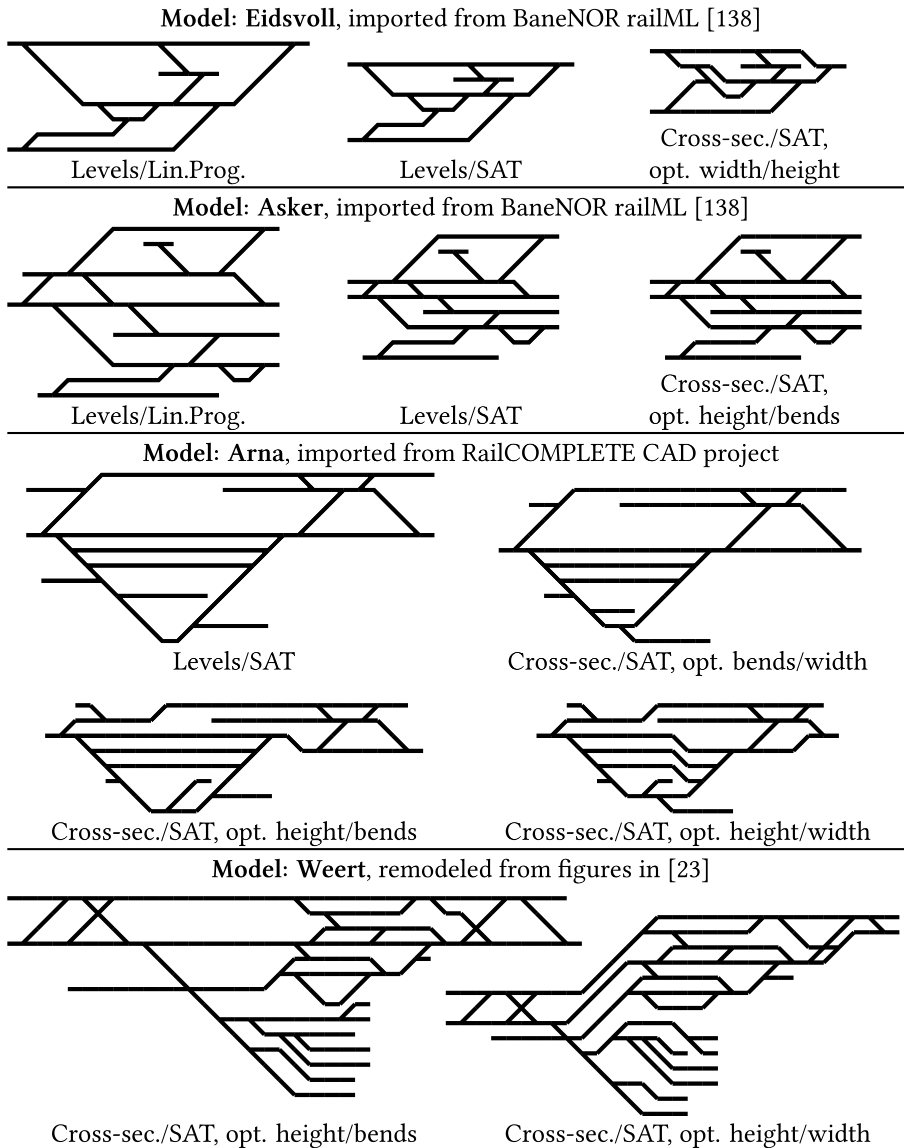


Figure 6.11: Comparison of three optimization models on various infrastructure models: Levels/Lin.Prog. (see Sec. 6.2.2), Levels/SAT (see Sec. 6.2.3), Cross-sec./SAT (see Sec. 6.2.4). Symbols and labels placed on the drawing may also affect layout (see Sec. 6.2.5).

Model	Src.	Size	Direct/SAT		Levels/SAT		Cross-sec./SAT			
			hwb	size (v/c)	bhw	size (v/c)	hwb	hbw	bhw	size (v/c)
Eidsvoll	[138]	35	60.7	57k/153k	0.02	2.3k/0.7k	0.05	0.06	0.33	4.0k/28k
Arna	RC	57	294	167k/493k	0.03	4.9k/1.3k	0.26	0.65	1.06	11k/100k
Asker	[138]	64	T/O	104k/295k	0.04	5.6k/2.0k	0.61	1.02	0.87	14k/124k
Weert	[23]	102	T/O	304k/969k	0.18	11k/4.0k	0.72	19.3	21.4	29k/327k
5x10	T	228	T/O	2.8M/13M	0.58	35k/2.7k	5.83	7.48	8.08	46k/364k
5x20	T	478	T/O	2.8M/12M	3.37	97k/7.7k	279	299	T/O	265k/4.2M
10x5	T	203	T/O	3.0M/14M	0.40	28k/2.0k	0.52	0.59	1.08	20k/83k
20x5	T	403	T/O	3.0M/14M	1.73	70k/4.0k	1.95	2.50	3.36	44k/165k
10x10	T	453	T/O	2.6M/12M	2.74	86k/5.5k	21.9	22.4	40.7	96k/727k
15x15	T	1053	T/O	2.3M/10M	22.7	255k/15k	T/O	T/O	T/O	N/A

Table 6.1: Running times in seconds on a mid-range workstation. Time-outs (T/O) indicate exceeding 300 s. Model sizes are given as the sum of the number of nodes and edges. Models were obtained from BaneNOR [138], a RailCOMPLETE CAD project (RC), and adapted from [23]. Scaling test models (T) named $n \times m$ consist of n serially connected stations, each spreading out to m parallel tracks. Optimization criteria are height (h), width (w) and bends (b). The size columns show the number of SAT variables and clauses (v/c).

6.3 Tool usage

A command-line tool that can generate the drawings as described in this paper is available online[1]. The tool can import railML files as track network input and track-side object symbols, or use a custom format for directly specifying topology. The tool offers two choices of built-in symbol appearances: "*simple*" for generic lamp-like signals and detector, and "*ERTMS*" for ERTMS-style marker boards and detectors (see the bottom part of Fig. 6.11). Extracting other object types from railML and producing other symbol styles can be done by post-processing JSON output, or by extending the tool using the Lua scripting language. The tool can produce output in JSON format (for custom visualization or post-processing), SVG (for use in web pages and web applications), or TikZ (for use in LaTeX documents).

We have implemented and compared the performance of the SAT-based methods described above, presented in Table 6.1 (the linear programming formulation is omitted for space, since it has lower quality output). We see that the Direct/SAT encoding has too poor performance to be of practical value. The Levels/SAT encoding is the fastest, and produces good output when optimizing for bends first. Cross-sec./SAT is slower, but is more capable for optimizing for height and width.

[1] https://github.com/luteberget/railplot

6.4 Conclusions and Future Work

We have demonstrated the feasibility of using an incremental SAT solver to automatically produce and optimize schematic railway drawings using several different optimization criteria. However, the choice of encoding makes a significant difference in the size of models that can be handled in a reasonable amount of time, cf. Table 6.1. The direct representation using an explicit grid fails to handle instances of relevant scale. Only after reformulating the problem in a more structured solution space, where the order of symbols is hard-coded into the problem, rather than added as a constraint after the fact, we were able to solve industrial-size instances in reasonable time for interactive use (i.e., under 1s).

Our goal is that professionals should be able to rely on high-quality automatic schematics, which requires further tailoring of symbol and text placement to specific use cases, and integration with GUI tools.

7 Integration with industrial end-user software

Civil engineering construction projects, such as railway projects, make heavy use of computer-aided design (CAD) tools to model the *geometric* aspects of the construction project and its product. The origins of CAD tools are in the computerizing of traditional *drafting*, which produces human-readable technical drawings that are used as plans and documentation for construction work. Mainstream CAD tools are mainly concerned with manipulating databases of geometrical objects constituting 2D or 3D representations of spatial properties, and the production of human-readable drawings which depict these geometrical structures.

The DWG file format created for the Autodesk AutoCAD software is a de facto standard in many engineering disciplines, and this format has also been adopted by several other CAD software packages.

Also, a new trend in construction projects is to use 3D models and visualization as a complement or an alternative to 2D drawings. In addition, such 3D modelling software packages often have features for *semantic* data, i.e. classifying different objects in the models and describing their properties according to some domain-specific catalogue of object classes, properties, and relations. 3D and semantic data are the main features of the building information management (BIM) programs, which are currently in the process of taking over as the next generation of CAD. There are currently no de facto standard BIM file formats, but the Industry Foundation Classes (IFC)[1] specification is a promising candidate.

This chapter first describes one approach to building a semantic CAD system by extending an existing 2D or 3D CAD system with railML-based railway classes, properties, and relations. We then show how the analysis techniques described in the chapters above have been (or are planned to be) integrated into a graphical user interface that railway engineers are already familiar with. This approach is the main architecture of the RailCOMPLETE software, based on the Autodesk AutoCAD software, where we intend to integrate the tools developed in this thesis.

[1]See the IFC Wiki: `http://www.ifcwiki.org/`

7.1 Semantic CAD

7.1.1 Grouping geometry into blocks

Grouping together several geometrical features into a single unit is in CAD terminology called "making a *block*". This allows the CAD user to create models more efficiently, by reusing commonly used components. The blocks, which may represent things such as chairs, doors, or railway signals, also create the opportunity to store higher-level information in a CAD model, other than the purely geometrical description. For example, if one uses a railway signal block to model a signal in a railway station, a program can count the number of signals in a model.

This idea can be extended by adding any number of attributes to a block. For a railway signal, we can add attributes that describe e.g. to which track it signals, along with its type, function, and direction. The CAD object database does then not only contain the geometrical objects, such as lines, curves, triangles, cubes, etc., but groups these primitives into higher-level concepts which are closer to the representation that one uses to reason about the actual working of the railway infrastructure.

With a good library of blocks (which we call a *symbol library*), the engineer can more efficiently build the geometric CAD models which lead to human-readable drawings, but they are also building a machine-readable model of high-level railway concepts. We call this *semantic CAD*. While this concept is also a part of building information modelling (BIM), BIM also includes many other concepts such as 3D visualization, time ("4D"), and cost ("5D").

The verification of signalling and interlocking rules requires information about properties and relations between objects such as which signals and signs are related to which track, and their identification, capabilities, and use. This information is better modelled by the railway-specific hierarchical object model *railML* [121]. In the CAD industry-standard DWG file format, each geometrical object in the database has an associated *extension dictionary*, where add-on programs may store any data related to the object. Our tool uses this method to store the railML fragments associated with each geometrical object or symbol, see Figure 7.1 . Thus, we can compile the complete railML representation of the station from the CAD model.

7.1.2 Object type descriptions

It is necessary to decide which objects in the CAD model should be associated with which data types, i.e. what attributes should be stored in the symbols. This is comparable to specifying an object's *class* in an object-oriented programming language. To do this, we create an *object type description* which augments the symbol library with class information. Whenever the user adds a symbol, its data editor is deter-

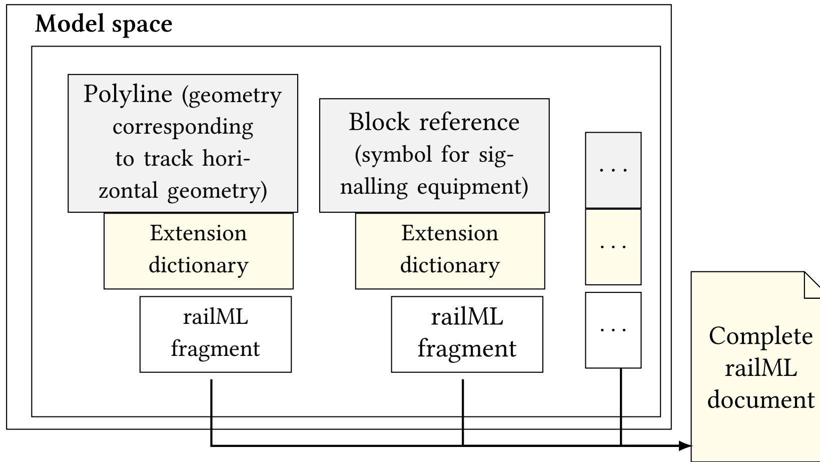


Figure 7.1: railML integrated into a CAD database

mined by the assigned class, and vice versa: when e.g. a railML object is imported into CAD, its corresponding symbol is inserted in the graphical model.

7.1.3 Interlocking and train protection systems

Besides the CAD model layout, the design of a railway station's signalling consists also of specifications for the interlocking and train protection (speed control) systems. These specifications are used to build the interlocking controllers and speed controllers, and they model the behaviour of the signalling equipment and its interaction with trains. These systems are tightly linked to the station layout.

A formal representation of the interlocking and train protection specifications is embedded in the CAD document in a similar way as for the railML infrastructure data, using the document's *global extension dictionary*. Thus, the single CAD document showing the human-readable, geographical layout of the train station also contains a machine-readable model which fully describes both the component layout and the functional specification of the interlocking and train protection systems. This allows analysis of the operational aspects of the train station directly in a familiar editable CAD model. See Figure 7.2 for an overview of this architecture.

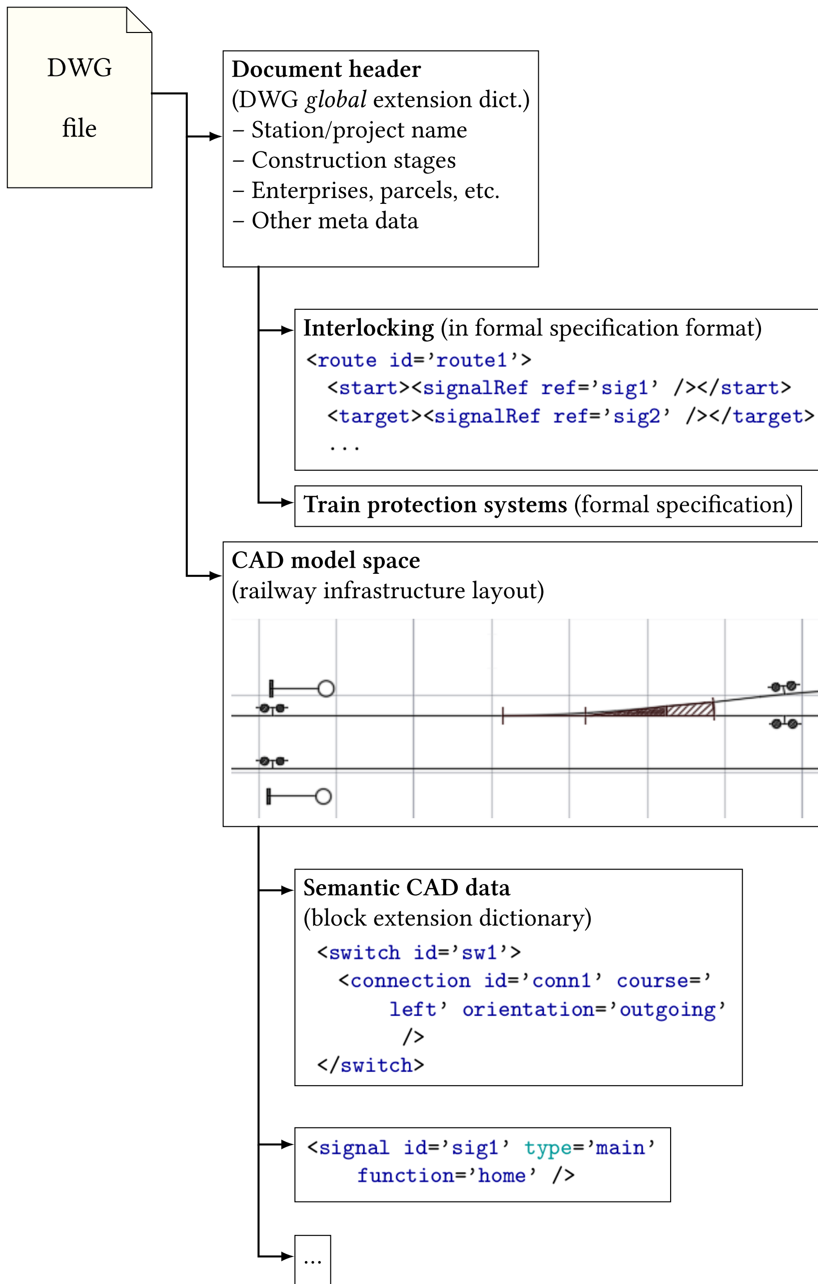


Figure 7.2: Semantic CAD document organization including interlocking specification.

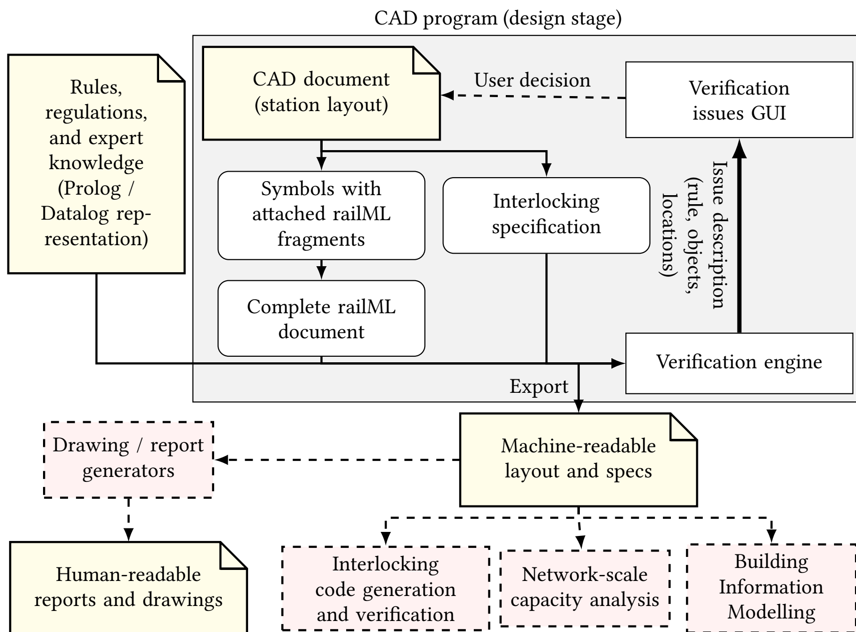


Figure 7.3: Railway design tool chain. The CAD program box shows features which are directly accessible at design time inside the CAD program, while the export creates machine-readable (or human-readable) documents which may be further analysed and verified by external software (shown in dashed boxes).

7.2 Railway analysis tasks as CAD plug-in programs

Figure 7.3 shows the overall architecture of a CAD program extended with on-the-fly verification of railway properties. The analysis tasks from the chapters above are integrated with the system in the following ways:

- **Static verification**, as described in Chapter 2, is well-suited for integration into a CAD program, especially when we want to edit the models (the railway infrastructure) and not the properties (Datalog code). Whenever the CAD model is opened or modified by the user, the railML model is compiled from the individual objects and sent to the verification program. Counter-examples are presented in a window containing a list of errors and warnings. See Figure 7.4.

  This type of verification can be compared to syntactic and static analysis of computer programs which is routinely used in integrated development environments for programming. In particular, our static verification is focused on integrating

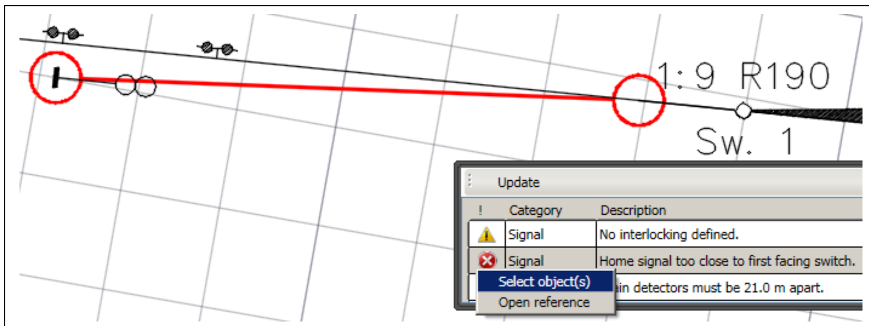


Figure 7.4: Counterexample presentation within an interactive CAD environment.

Generelle krav

Utførelse

På jernbaneskilt er det naturlig å skille mellom høyest mulig og en lavere refleksevne. Dette fremgår av tegningen for det enkelte skilt.

For å unngå speilrefleks, bør skilt og merker ikke settes opp vinkelrett (90º) på sporet, men dreies 4º ut.

ID: skilt1 — Definisjon.

RailCNL: En skilt har refleksevne høy eller lav.

AST: Constraint (SubjectClass (StringClass "skilt")) (ConditionPropertyRestriction (MkPropertyRestriction (StringProperty "refleksevne") (OrRestr (Eq (MkValue (StringTerm "høy"))) (Eq (MkValue (StringTerm "lav"))))))

ID: skilt2 — Automatisk verifisering.

RailCNL: En skilt bør ha sporvinkel som er større enn 94.

Datalog:

- skilt2_found(Subj0) :- skilt(Subj0), sporvinkel(Subj0, Val2), Val2 > 94.
- skilt2_recommendation(Subj0) :- skilt(Subj0), !skilt2_found(Subj0).

Figure 7.5: Paraphrasing view which shows original regulations and their translation into controlled natural language side by side. From the counter-example presentation (see Figure 7.4), the user can request to see the source of the rule which reported the warning or error and in this way check that the specifications are correct and also learn more about the specifications.

and automating those simple, yet tedious, rules and conditions usually used to maintain some form of consistency of the railway, and have these checks done automatically.

- **Regulations tracing** can be a useful tool for investigating the cause of a warning or error that has been reported from a verification program. Using the approach described in Section 5.5 in the context of controlled natural language specifications, engineers can trace the translation steps from the original regulations texts through the Datalog representation and into their geographical model of the railway infrastructure. See Figure 7.5 for an example in Norwegian language.
- **Schematic plans**, as described in Section 6.1, are often preferred by railway engineers when working with interlocking specifications and dynamic evaluations of operational scenarios. Maintaining corresponding geographical and schematic plans automatically as complementary CAD models is a conceivable approach for the schematic visualization algorithm described in Chapter 6. However, in order to better support presentation and interaction with executions of operational scenarios in the context of local capacity verification, we have prototyped a separate visualization tool with schematic presentation which presents the output of capacity verification. See Figure 7.6 and Figure 7.7 for screen shots from this separate tool, which presents a time choice slider, a schematic view with trains current position, and a time/distance diagram which presents the whole timeline, similar to a blocking time diagram (see Figure 4.1).
- **Capacity verification**, as described in Chapter 3, requires two extensions to an infrastructure editor: (1) presentation of operational scenario execution timelines where trains' positions are overlaid on a drawing of the infrastructure and (2) an editor for capacity specifications (defined in Section 3.3). The tool in Figure 7.6 currently supports the former but not the latter, and also does not have interactive editing of the infrastructure itself. Both of these features could be added to a 2D geographical CAD program, and we plan to experiment with this in the RailCOMPLETE editor.
- **Synthesis**, as described in Chapter 4, requires some fundamental changes to the usual editor paradigm for infrastructure: trackside objects can now be either *fixed*, meaning that it is specified by the user and should not be moved or removed by the synthesis algorithm, or *derived*, meaning that it has been suggested by the synthesis algorithm and can be made fixed. The user should be able to turn the synthesis to *fully automatic*, where objects are added automatically to fulfil specifications, or to *suggestion mode*, where individual optimization edits are suggested to the user. We have prototyped a schematic-only editor with these features, see Figure 4.11 on page 88. Because CAD plug-in programs are modular parts in an already comprehensive software package, they should not interfere with the basic editor user interface, and it is not immediately evident

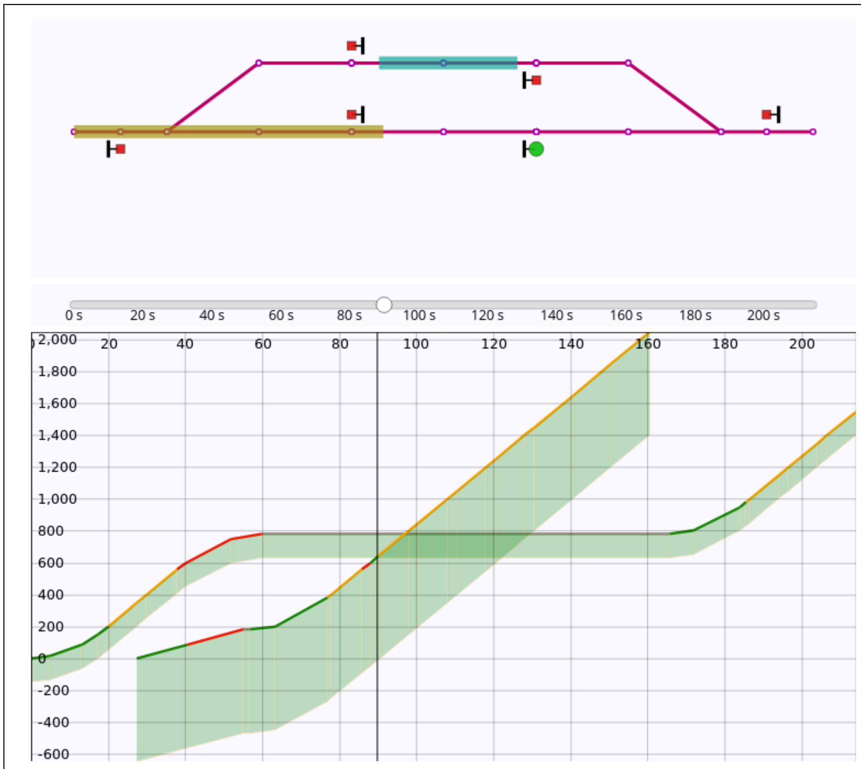


Figure 7.6: Simple example of schematic and time/distance diagram used to visualize a sequence of events in an operational scenario.

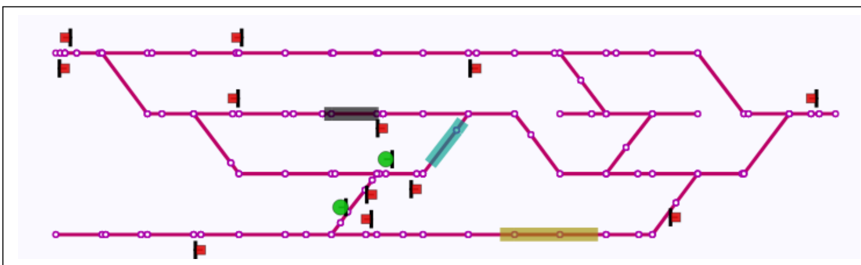


Figure 7.7: A realistic example of an operational scenario involving multiple trains at Eidsvoll station.

how this editing paradigm should be combined with the CAD program. We plan to investigate how these tools can be suitably combined.

There are several other analysis tasks that are relevant for signalling engineering that we have not integrated here, such as interlocking implementation development, or network-scale capacity analysis. For other railway engineering subdisciplines there are also domain-specific analysis tasks, such as structural analysis, electrical analysis, and many more.

The input data and editor paradigm for each one of these analysis tasks needs to be carefully considered to decide whether they are suitable for integration with infrastructure editors (schematic or geographic), or more suited to keep as a fully separate program.

7.3 Other external analysis tools

Generally, analysis and verification tools for railway signalling designs can have other complex inputs in addition to the tracks and trackside objects which have been the main subjects of this thesis. Also, other analysis tools must account for a large variety of situations, and they usually require long running times. Therefore, we by default limit the verification inside the design environment to static rules and expert knowledge, as these rules require less dynamic information (timetables, rolling stock, etc.) and less computational effort, while still offering valuable insights. This situation may be compared to the tool chain for writing computer programs. Static analysis can be used at the detailed design stage (writing the code), but can only verify a limited set of properties. It cannot fully replace testing, simulation and other types of analysis, and must as such be seen as a part of a larger tool chain.

Other tools, that are external to the CAD environment, may be used for other types of analysis, which are less automated or require heavier computation, such as:

- **Code generation and verification for interlockings** is possible e.g. through the formal verification framework of Prover Technology[2].

  Railway infrastructure topology, signalling objects, and interlocking specifications should be automatically transferred to a code generation and verification tool to help automate interlocking implementation. The transfer of data from the CAD design model to interlocking code generation tools is possible by using standardized formats such as railML, which recently also incorporated an interlocking specification schema.

[2]Prover Technology AB: `http://www.prover.com/`

- **Capacity analysis and timetabling** can be performed using e.g. OpenTrack[3], LUKS[4], or Treno[5].

  OpenTrack is a simulation tool which allows stochastic capacity analysis, running time analysis, and other types of analyses. By transferring data directly from a CAD model, such analyses can be performed at an early stage in the design process, greatly increasing the possibility for design decisions to be affected by capacity analysis. This allows a more agile and dynamic design process, so that the end goals of the railway administration can be met, and costs of re-designing and re-building can be minimized.

- **Building information modelling (BIM)**, including such activities as life-cycle information management and 3D viewing, are already well integrated with CAD, and can be seen as an extension of CAD.

  The object type definitions described in Section 7.1 above may be used to associate 3D models to symbols in the 2D geographical layout. Semantic information can then be preserved when transferring information between 2D and 3D representations. 3D tools for design and presentation are now becoming widely used on new railway projects.[6]

  In the future, it is likely that 3D BIM models will largely replace 2D CAD models, but as of today it is a reasonable approach to compile 3D models from a 2D model with semantic information.

---

[3] OpenTrack: simulation of railway networks, `http://www.opentrack.ch/`

[4] LUKS: analysis of lines and junctions, `http://www.via-con.de/en/development/luks`

[5] treno: timetable reliability & network operations analyser, University of Trieste.

[6] `http://www.jernbaneverket.no/Prosjekter/Inter-City-/3d/`

8 Conclusions

In the chapters above, we have combined automatic reasoning and formal specifications in a way that solves both general and specific problems that are relevant for railway planning and engineering. One of the goals that we set in Section 1.1 was to use formal specifications so that maintenance and handling of complexity could be assisted by automated tools. This style of software development has the potential to overcome some of the problems with developing software tools for railway engineering, mainly that the regulations are complex, and that they vary in both major and subtle ways between countries and administrations. We conclude this thesis in the sections below with some perspectives on these goals.

8.1 Generality, maintenance, and user-friendliness

The technique of using Datalog logic programming (Ch. 2) for static analysis is the most general analysis form of the ones we have investigated. Because static analysis of railML documents is concerned with writing properties that refer more or less directly to the structure of the input document itself, the procedures for deciding whether the properties have been satisfied are tractable and have often little impact on the performance of an interactive editor. Instead, the flexibility and ease of specifying properties is more important for practical use. Datalog has such flexibility, and in order to create our CAD-based static verification tool, we did not have to do much programming of business logic, outside of a trivial translation from railML to Datalog facts, and a base library of Datalog railway properties. This also means that, from a railway engineering point of view, there is not much maintenance to be performed on the *inside* of the verification system itself, and engineers and software developers have mostly the same perspective on the system: a simple logic system taking complex, composable specifications as input. Datalog has similarly been demonstrated as a concise and elegant way of solving static analysis of computer programs. With recent developments in solver techniques, Datalog program analysis can be as fast as special-purpose algorithms.

The user-friendliness of logic programming can certainly be debated, and the idea of expecting railway engineers to master a somewhat esoteric programming language paradigm is not clearly feasible. The controlled natural language system that we have developed (Ch. 5) mainly brings the specification of static properties into a more user-friendly language and framework. However, some of the simplicity and generality of Datalog logic programming is also lost along the way. Railway-specific language constructs were necessary to write relevant properties naturally and concisely. We only integrated the RailCNL front-end language with

the Datalog-based static verification system, but one advantage of using a domain-specific front-end language is that it could also have had the capability to recognize which logic domains were relevant for a given linguistic construct. For example, statements about capacity properties could be translated into the dynamic verification system (Ch. 3) and solved by the corresponding solver method.

Although static properties are certainly a major source of every-day drudgery for railway engineers, the adjustments required to fulfil statically checkable rules and regulations often have only a shallow impact on the actual operation of a finished railway system. The important major decisions in such a system are the ones which impact the dynamic behaviour of trains travelling on the infrastructure, and these decisions can only be taken by considering a large set of different dynamic situations. The dynamic verification system described in Ch. 3 performs a more specific task than the static verification system, but the task itself is a central and important question in railway signalling engineering. Our approach does, however, suffer, like all similar capacity analysis packages that we are aware of, from having to model operational behaviour as code ("*hard-coded*" behaviour), which is more coupled and brittle than logic-based specifications. This makes it complicated and risky to let the user have full control over it and to change it while performing analysis tasks. Compared to the Datalog system, this system has much more complexity on the *inside*, and there is a divide between the software developer's view of the system and the end-user railway engineer's view. One attempt to tackle this problem by Phillip James et al. (see [80]) was written the Maude rewriting logic system. They use a hybrid approach of model checking and simulation, much like our dynamic verification system, but in the Maude approach, both parts are written in the same language. However, it is not, in our opinion, more user-friendly than editing simulation logic in a general-purpose imperative language.

Finally, drawing schematic plans of railway infrastructure (Ch. 6) are the most specific problem we have tackled in this thesis. It is not so much an example of formal methods as it is a demonstration of problem-solving and separation of concerns by performing mathematical modelling and calling a *solver*, i.e. a program that solves a mathematical problem. SAT solvers form the underpinnings of many formal methods-related tools such as model checkers [14], but are also good basic algorithms in their own right. It can be surprising and humbling to try to come up with a clever special-purpose algorithm and then get beaten, in terms of both performance and elegance, by a much simpler implementation by translation to SAT.

8.2 Design automation, quality, and standards compliance for tools

We have used techniques from computer science, logic, and formal methods to develop railway engineering software tools, but note that the goal has not been to per-

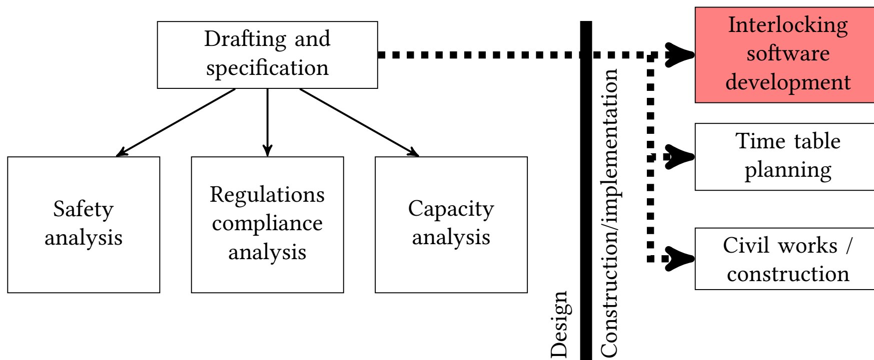


Figure 8.1: Process of railway engineering with manual control barrier between the design phase and the construction/implementation phase. The red box with interlocking software development is safety-critical and cannot trust inputs, such as interlocking specifications, coming from non-verified software.

form formally verified software development, but creating analysis tools that help efficiently create a railway infrastructure design that is of high quality and gives the intended traffic capacity and is realizable from the civil construction works' point of view. Only later in the process comes formally verified control system software development which results in the safety critical interlocking, and is not in the scope of this thesis. See Figure 8.1.

However, whenever automated tools are introduced to a task that has been dominated by manual work, there is usually a tendency for the manual control processes to become weaker, especially if the automated tool seems to perform the task correctly in most cases – manual control standards will be slipping. To avoid quality issues carrying over to safety-critical systems, there are strict standards for using automated tools in development of such control software. Specifically, the EN 50128 standard for developing safety-critical software for railway applications classifies development tools into the following categories:

- T1 generates no output that is able to contribute, directly or indirectly, to the executable code (including the data) for the safety-related system
- T2 takes care of the testing or verification of the design or the executable code, when errors in the tool itself may prevent it from detecting faults, but cannot directly create any errors in the executable software;
- T3 generates outputs which are able to contribute, directly or indirectly, to the executable code for the safety-related system.

As the traditional barriers of manual control are in place between the design and the construction/implementation phase, where quality control at the hand-

over is performed by a human, the tools presented in this thesis can be used in the design phase by each railway engineer individually or in collaborations in a railway engineering organization. However, if the data and results in these tools are directly transferred and used in the implementation of software, then the qualification requirements of EN 50128 should apply, classifying them as development tool of type T2.

8.3 Railway engineering in the future

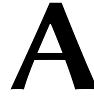
The big new trend in construction projects, also in railway construction, is the use of building information modelling (BIM) software for data exchange, collaboration and 3D visualization. BIM tools have the potential to improve the engineering profession in terms of cross-discipline communication and large-scale project management. However, the deeply specialized and knowledge-intensive analysis tasks that are performed within each engineering sub-discipline are typically only supported by software for the tasks that have the highest impact on cost, since the cost of tailor-made, process-integrated software development is high. These tasks are especially demanding and important in railway systems, where central control makes the various parts of the design highly coupled.

Standard exchange formats for railway information, such as railML, and also IFC Rail, which is part of the larger IFC BIM data system, have the potential to spur development of analysis software in a different style. Engineers who are interested in programming, or small development companies who are interested in railway engineering, can develop more minimalist, modular programs which do the analysis tasks. In contrast with the centralized hierarchical organization and process-oriented world of BIM software from the big players, we believe that the state of railway engineering software tools can benefit from more specialized software developed in the style of the *Unix philosophy* (see [146]). With the current state of the art in solver programs (which are themselves often minimalist and modular), it is feasible to solve hard numerical and combinatorial problems, and to maintain them and change them in face of changing official requirements, without large research and development teams. The resulting tools are of course more limited in their capabilities and in their user interfaces than a comprehensive, expensive development project, but with good standards for data exchange, complemented by industry-standard heavy-duty tools for visualization and project management, we believe that the small-scale, composable and modular approach has the potential to create better special-purpose analysis tools.

The introduction of the ERTMS/ETCS standard in signalling has the potential to become an international de facto standard, which means that regulations and practices will become more uniform across different countries. This is a great opportunity for the international railway engineering community to unite around better teaching materials, technical documentation, best practices, and analysis software.

Such materials are often not viable as commercial products, and should instead be shared for everyone's benefit, giving increased safety and performance for all railways around the world.

Junction manual



The following pages consitute the user manual for the Junction tool as of September 2019. Junction was developed in 2019 as a proof of concept for a more user friendly end-user tool demonstrating the results presented in Chapters 3, 4, and 6 of this thesis. The program was developed using the Rust programming language[1], using OpenGL and ImGUI[2] for graphics and user interface, and using railway analysis features from the library and command line programs produced during the work on this thesis. The program is available for download from:

`https://luteberget.github.io/junction`.

[1] See `https://www.rust-lang.org/`
[2] See `https://github.com/ocornut/imgui`

A.1 Overview

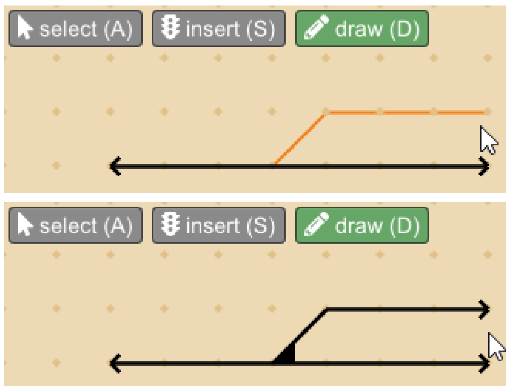
Download (Win/Linux/MacOS) Repository

Junction is a railway operations analysis tool for small-scale infrastructure, such as construction projects. It focuses on quickly building or importing infrastruture models, and then letting you dispatch trains to examine capacity properties of the track layout and signaling equipment. You can manage several dispatch scenarios and see how their timelines change when you make changes to the infrastructure. Junction also features an auto-dispatch mode, where you supply a high-level description of operations, and the program works out the required dispatch commands needed to execute the operations, resulting in a set of test cases for your signaling design that help you when making changes as your project progresses.

Quick tour

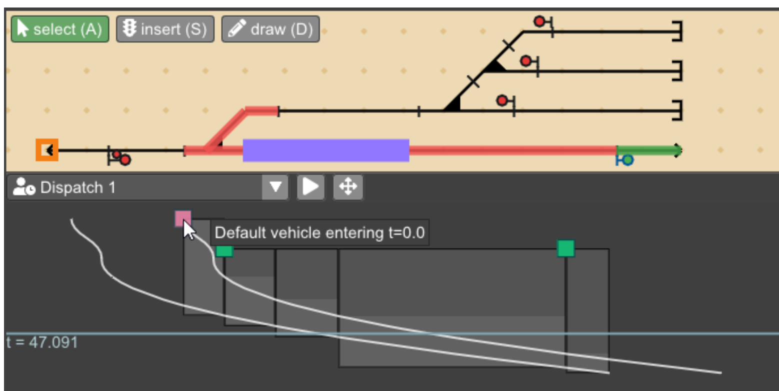
- Quickly build the **infrastructure** of tracks and signaling equipment by drawing lines on a grid. Switches and crossings are automatically identified and displayed based on the lines.

  See Infrastructure.

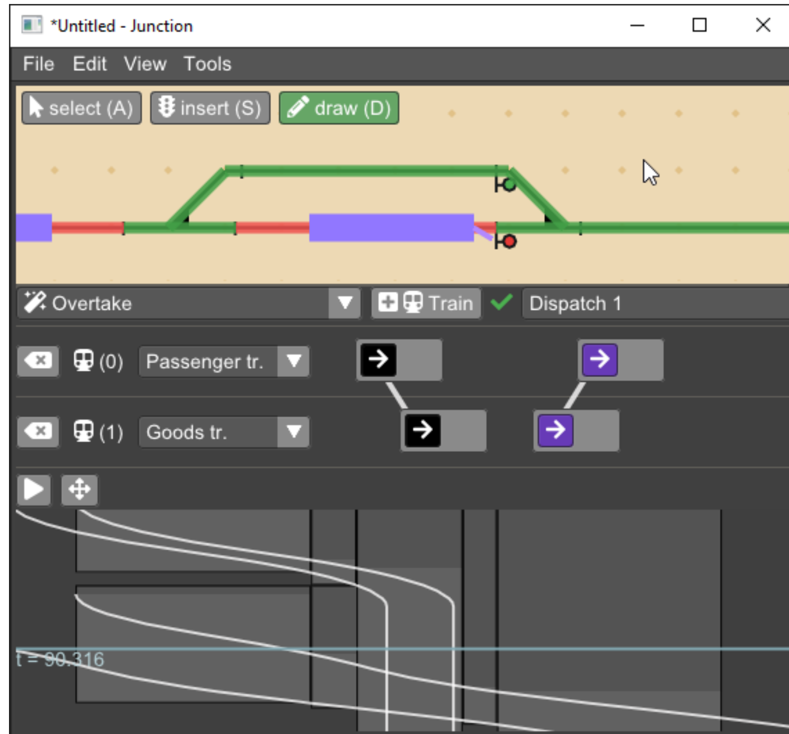
- **Dispatch** individual trains using train routes by pointing to the starting location of a train route and selecting a route from the menu.

  See Dispatching.

- Build **plans** representing train operations such as crossing, overtaking, train frequency, etc., and get a list of possible dispatch patterns that satisfy the plans. When you continue adjusting the infrastructure, the plans will be updated and you can check at a glance that operations are still working.

  See Planning.



A.2 Getting started

Installing

You can install Junction by downloading a binary executable from here:

- github.com/luteberget/junction/releases/latest

There is no installation program, only a single executable that you can put wherever you want.

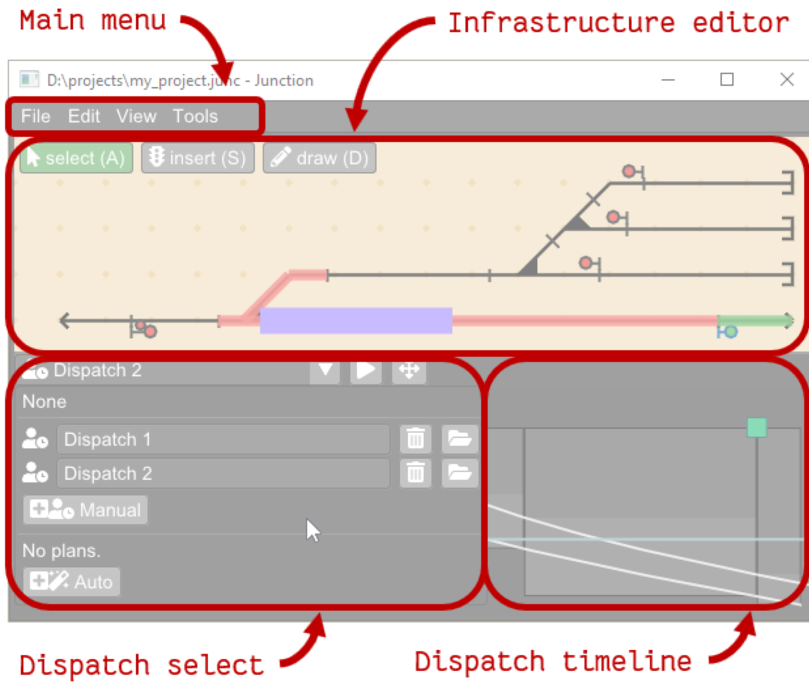
Building from source

If you would like to build Junction from source or modify the program, the source code can be downloaded from the Github repository at github.com/luteberget/junction. Building the project depends on the Rust compiler toolchain and a C++ compiler toolchain being installed on your system.

Usage

The main window consists of the following components:

- The main menu bar (see Main menu).
- The infrastructure editor (see Infrastructure).
- The dispatch selection menu (see Dispatch).
- The dispatch output diagram (see Dispatch).



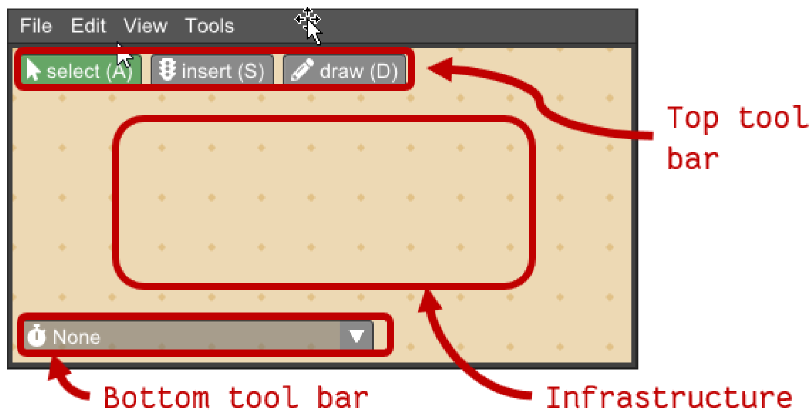
Main menu

The main menu let you load, save, import, and export documents, and lets you open the following tool windows:

- File
  - Import/export railML files.
- Edit
  - Edit vehicles (see Vehicles).
  - Signal designer (see Signal designer).
- View
  - Log view (see Log).
  - Model inspector (see Model inspector).
  - Configure settings (see Settings).

A.3 Infrastructure

The top part of the Junction main window shows the infrastructure, consisting of tracks, track nodes, and signalling objects. When Junction is first started, or a new document is created, the infrastructure is empty and the window shows only a blank canvas and buttons for each editing tool.



Top tool bar

The top tool bar has the following tools:

-

Select items (hotkey ‘A’).

  Left-click on tracks, nodes, or objects to select them. Currently selected items are highlighted by color. Hold the Shift button to add to selection instead of replacing. Left-click and drag while the pointer is not over any item to draw a selection window. When releasing the mouse button, all items side the selection window will be selected. Left-click and drag while the pointer is over an item to move the item. If the item is part of the current selection, all currently selected items will be moved.

-

Insert object (hotkey ‘S’).

  Insert railway signalling objects. The available objects are:

  - Main signal (with or without distant signal)
  - Detector (train vacancy detection section boundary)

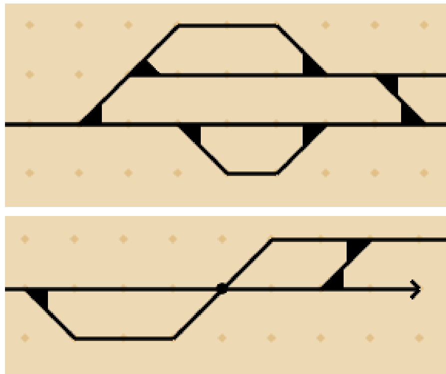
  When inserting a main signal, click on a location beside a track to insert the signal at that side of the track. The side of the track determines the travel direction the signal faces. Trains see signals on the right-hand side of the track.

  Detectors are placed in the middle of the track, and constitute a section boundary for a train vacancy detection section.

-

Draw tracks (hotkey ‘D’).

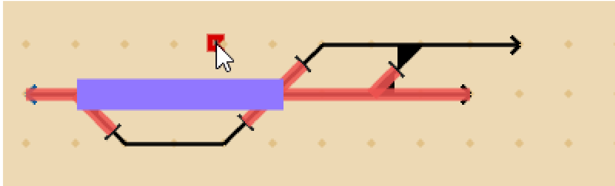
  Left-click and drag to draw lines representing tracks onto the infrastructure grid. Whenever more than two lines meet at a grid point, a node type is detected and a node symbol is displayed. Three lines meeting will produce a switch, and four lines meeting will produce a crossing. Examples:

Bottom tool bar

The bottom tool bar contains the closed version of the dispatch select menu. Click the tool bar to open the dispatch select menu. See Dispatch.

When a dispatch is opened, the current state of the infrastructure at the time selected in the diagram view (see Dispatch) is shown by graphics overlaid on the infrastructure view. These graphics show e.g. train positions and detection section status, but do not affect the use of the infrastructure editor. You may still edit the infrastructure while the dispatch state is shown, and the state will be updated accordingly.



Context menu

The context menu is opened by right-clicking on the infrastructure view. If the pointer is currently over an item, the selection will be set to that item before opening the context menu. The context menu contains actions that are relevant to the currently selected items:

- Delete, works for all items. When deleting a node, only the changes from the default detected node are deleted, as the number of lines meeting at a grid point will still determine the node type.
- On nodes and objects, the available properties of that node or object type are displayed in the context menu. For example, crossings may have type (a) crossovers (no switching), (b) single slip (switching in one direction) or (c) double slip (switching in both directions). Main signals may have distant signals enabled or disabled.
- On boundary nodes and main signals, available train routes are shown. If you click a train route, it is added to the current active dispatch at the current time. If no dispatch is active, a new dispatch will be created and opened, and the selected train route will be added to it. See Dispatch.
- On all items, adding the item as a visit location in the currently active plan is shown. If no plan is currently active, a new plan is opened, a new train is added to it, and the visit location is added to the train. See Planning.

A.4 Dispatch

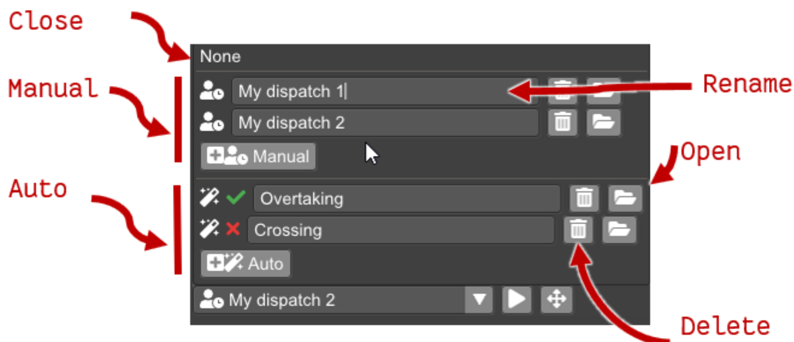
From the infrastructure view, you can start dispatching trains by either:

1. Right-clicking a boundary node and starting a train from there. If no dispatch is currectly selected in the dispatch selection menu, a new dispatch will be created.
2. Opening the dispatch selection menu and adding a new dispatch.

The dispatch selection menu is used to open or close the dispatch view, to switch to another dispatch or to add, delete or rename dispatches.

Dispatch selection

The dispatch selection menu is always located at the bottom of the infrastructure view.



The dispatch selection menu presents the dispatching **modes**:

- **None**: closes the dispatch view, leaving only the infrastructure.
- **Manual dispatches**: dispatches where you can add and remove commands.

  Commands are either:

  - **Trains**: a train of a given vehicle type appears in the model through a specified boundary traintrain route.
  - **Route**: a train route is activated.

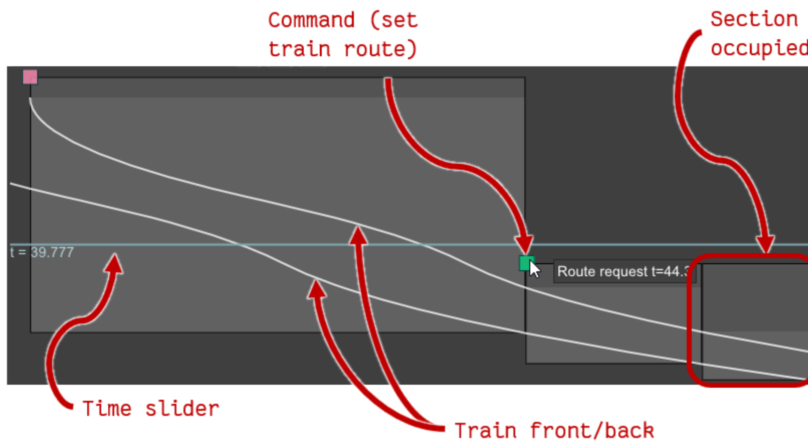
- **Auto dispatch**: plan dispatched by giving constraints for train movements, and see a list of possible dispatches for the plan on the current infrastructure. See Planning.

The dispatch selection menu has buttons for adding new dispatches of each mode, renaming, deleting, and opening them. Opening a dispatch opens the dispatch diagram and planning view (if an auto dispatch is selected). Each auto dispatch also has an icon showing whether it is currently satisfied or not on the current infrastructure.

Dispatch diagram view

When a dispatch has been opened in the dispatch selection menu, a diagram is shown at the bottom part of the window. The diagram is a time/distance diagram with time on the vertical axis and distance (mileage) on the horizontal axis. The diagram contains:

- Squares representing the commands in the dispatch. Right-clicking a command brings up a context menu where the command can be deleted. Left-clicking and dragging the command adjusts the time that the command is given.
- Boxes representing the train vacancy detection status of each section.
- Curves representing the front and the back of each train.



Interaction in the dispatch view

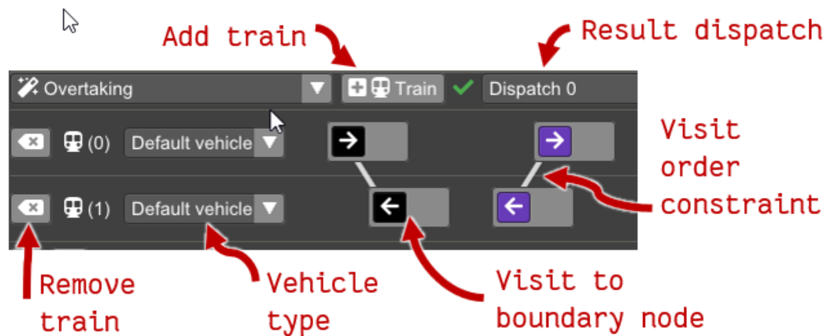
Left-clicking the diagram view will set the time slider to the time corresponding to the vertical position of the cursor.

If the currently opened dispatch is a manual dispatch, the dispatch can be edited.

New commands are added to the dispatch from the infrastructure context menu (see Infrastructure).

A.5 Planning

Whenever an **auto dispatch** is selected in the dispatch selection menu (see Dispatch), the planning menu is opened.



The planning view lets you edit the plan and view all the alternative dispatches that satisfy the constraints.

- **Add trains** button adds a new row to the bottom of the planning area, representing a new train with no locations it must visit. Visits can be added to the train by right-clicking in the infrastructure view and using the context menu to add a visit to that location to a train.
- **Result dispatch** menu shows whether any dispatches were found, showing a green checkmark for success or a red cross for fail. Selecting a dispatch from the menu brings up the diagram view which is used in the same way as the diagram for the manual dispatch, except that the commands can not be edited.
- **Trains** are shown one in each row of the planning view. Trains can be removed and their vehicle type can be set. Visits can be added to the train by right-clicking in the infrastructure view and using the context menu to add a visit to that location to a train.

  Visits are used as follows:

  - Each visit is shown as a box on the train's planning row. A train must go to any of the locations inside each of its visits.
  - Left-clicking and dragging a visit between other visits (on the same or on another train) moves the visit. Dragging a visit onto another visit, merges the

locations in the dragged visit into the target visit. The train must then visit at least one of the locations in the visit.

- Hovering the mouse over a visit or a location inside a visit, highlights the location in the infrastructure view.
- Right-clicking a location brings up a context menu where visits, locations, or constraints may be deleted.
- The context menu also allows adding new constraints. After clicking the add constraint menu item, a line will be shown between the clicked visit and the mouse pointer. If you click on another visit this adds a planning constraint that the first visit must come before the second visit. All constraints are shown as lines between the visits.

A.6 Tool windows

In addition to the main interface (see Getting started), the following tools are available as separate windows which can be opened from the main menu.

- Edit vehicles (see Vehicles).
- Signal designer (see Signal designer).
- Log view (see Log).
- Model inspector (see Model inspector).
- Configure settings (see Settings).

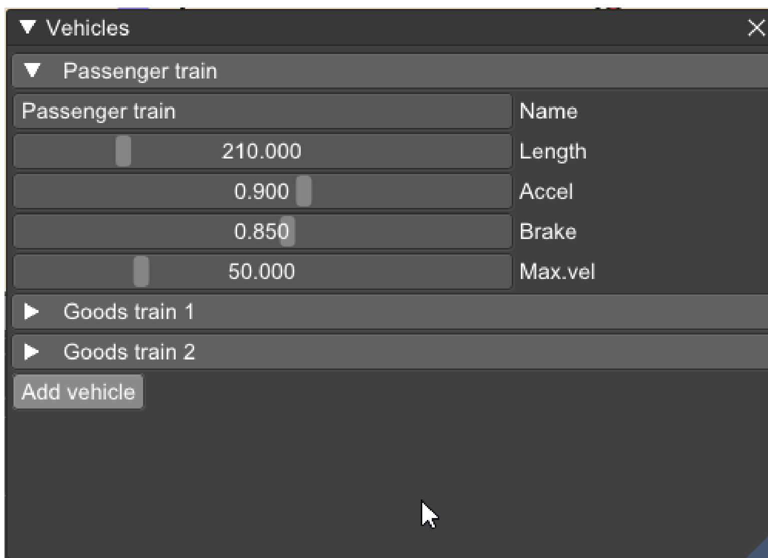
TABLE OF CONTENTS

- Vehicles
- Signal designer
- Log
- Model inspector
- Settings

A.6.1 Vehicles

Vehicles used in dispatches and planning are defined by the following characteristics:

- Name
- Length
- Maximum velocity
- Maximum acceleration
- Maximum braking de-acceleration

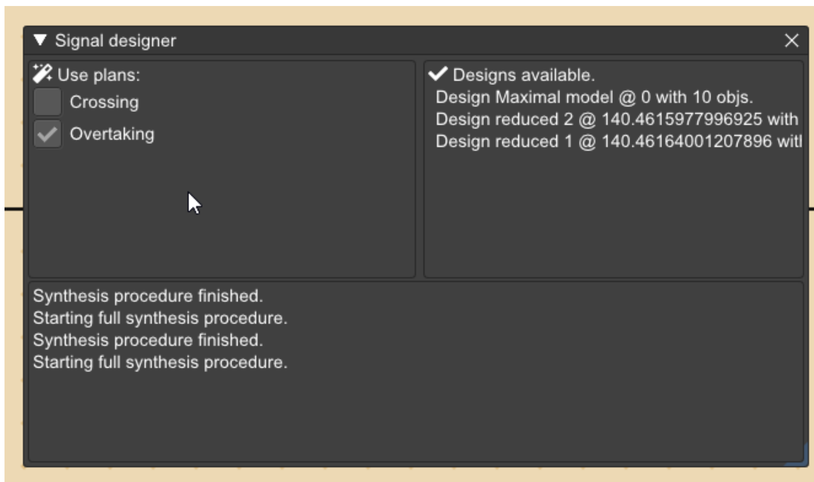


A.6.2 Signal designer

The signal designer can be used to synthesise a signalling design given an infrastructure containing tracks but no signals, and given a set of auto-dispatch specifications (see Dispatch).

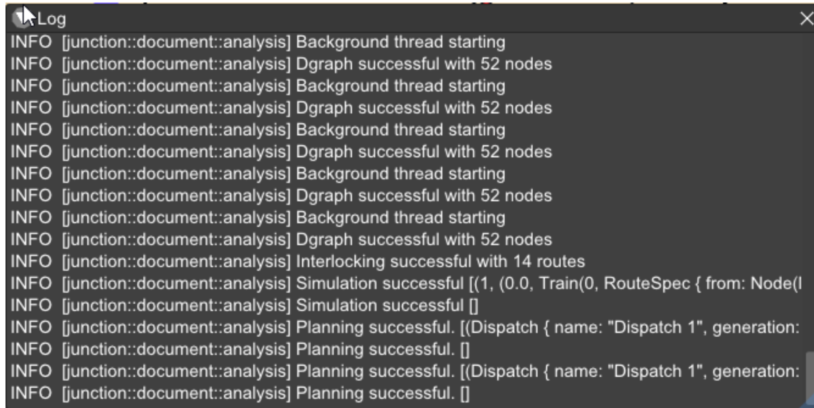
The set of auto dispatches to be used as the basis for the synthesis are shown in the left part of the window, and can be deactivated if you do not wish for the design to satisfy this auto dispatch plan.

The right part of the window contains the designs that are output from the synthesis procedure, along with their score. Select a design from the list to add it to the current infrastructure.



A.6.3 Log

The log window shows diagnostic output from all modules of the Junction program.



A.6.4 Model inspector

The model inspector shows you a tree representation of the analysis model that is being edited, and also all of the derived information about the model that is used for dispatching.

The model itself consists of:

- Line segments representing tracks.
- Node properties, overriding the defaults.
- Objects locations and functions (main signal, detector, etc.)
- Vehicle type specifications.
- Dispatch specifications.
- Plan specifications.

The analysis output consists of:

- Topology: the inferred track and node types gathered from the line segments.
- DGraph: the railway network double-node graph used for simulation.
- Interlocking: elementary routes gathered from consecutive main signals and the detection sections that the train must pass over to get from the begin signal to the end signal.
- Dispatches: one simulation output for each manual dispatch, providing the timeline used for the dispatch diagram view. For each plan specification (auto dispatch), there is a (possibly empty) set of simulation outputs.

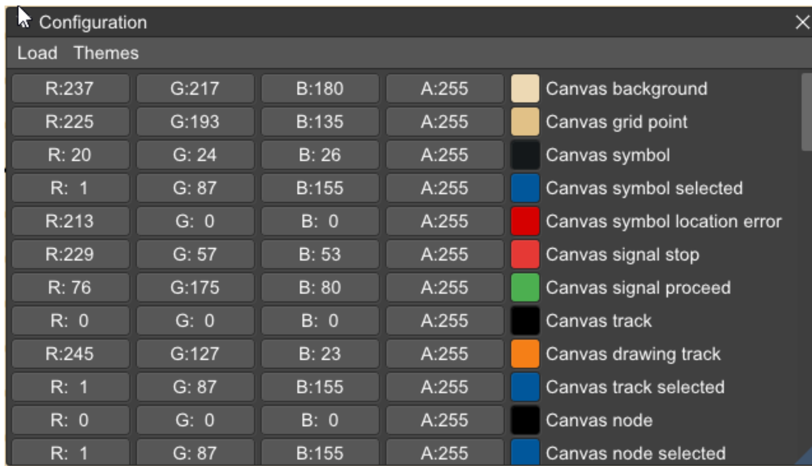
View data

- Model
  - Line segments
  - Objects
  - Node data override
  - Vehicles
  - Dispatches
  - Plans
- Derived data / view model
  - Topology
  - DGraph

```
Some(
    (
        2,
        DGraph {
            rolling_inf: StaticInfrastructure {
                nodes: [
                    Node {
                        other_node: 1,
                        edges: ModelBoundary,
                        objects: [
                            3,
                        ],
                    },
                    Node {
                        other_node: 0,
                        edges: Single(
                            2,
```

A.6.5 Settings

The settings window lets you change colors and drawing style. Preset themes are available from the menu bar.



Bibliography

[1] IRS 30100: RailTopoModel - railway infrastructure topological model. The International Union of Railways (UIC), 2016.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1995.

[3] M. Abril, F. Barber, L. Ingolotti, M.A. Salido, P. Tormos, and A. Lova. An assessment of railway capacity. *Transportation Research Part E: Logistics and Transportation Review*, 44(5):774 – 806, 2008.

[4] Martin Agfjord, Krasimir Angelov, Per Fredelius, and Svetoslav Marinov. Grammar-based suggestion engine with keyword search. In *Proceedings of The Fifth Swedish Language Technology Conference (SLTC 2014), Uppsala, Sweden*, 2014.

[5] Krasimir Angelov, John J. Camilleri, and Gerardo Schneider. A Framework for Conflict Analysis of Normative Texts Written in Controlled Natural Language. *JLAP*, 82(5-7):216–240, 2013.

[6] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *The 2015 ACM SIGMOD International Conference on Management of Data*, pages 1371–1382. ACM, 2015.

[7] Gilles Audemard and Laurent Simon. On the glucose SAT solver. *International Journal on Artificial Intelligence Tools*, 27(1):1–25, 2018.

[8] Silvania Avelar. *Schematic Maps on Demand - Design, Modeling and Visualization*. PhD thesis, ETH Zürich, 2002.

[9] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[10] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[11] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Springer, 2007.

[12] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, Yunshan Zhu, et al. Bounded model checking. *Advances in computers*, 58(11):117–148, 2003.

[13] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, TACAS '99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[14] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[15] Magnus Björk. Successful SAT encoding techniques. *JSAT*, 7(4):189–201, 2011.

[16] D. Bjørner. New results and trends in formal techniques for the development of software in transportation systems. In G. Tarnai and E. Schnieder, editors, *Proceedings of the Symposium on Formal Methods for Railway Operation and Control Systems (FORMS'03)*, pages 1–20. L'Harmattan Hongrie, 2003.

[17] Ernst-Josef Blum. *ROSY – Menü-basiertes Parsing natürlicher Sprache unter besonderer Berücksichtigung des Deutschen*. Diploma thesis, FB. Informatik, University of Saarland, Saarbrücken, 1987.

[18] Paul P. Boca, Jonathan P. Bowen, and Jawed I. Siddiqi. *Formal Methods — State of the Art and New Directions*. Springer-Verlag, 2010.

[19] A. Borälv and G. Stålmarck. Prover technology in railways. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Industrial-Strength Formal Methods*, International Series in Formal Methods, pages 329–305. Springer-Verlag, 1999.

[20] Arne Borälv and Gunnar Stålmarck. Formal verification in railways. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Industrial-Strength Formal Methods in Practice*, pages 329–350. Springer, 1999.

[21] Mark Bosschaart, Egidio Quaglietta, Bob Janssen, and Rob M. P. Goverde. Efficient formalization of railway interlocking data in RailML. *Information Systems*, 49:126–141, 2015.

[22] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi A. Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *LNCS*, pages 317–333. Springer, 2005.

[23] Angela Brands. Automatic generation of schematic diagrams of the Dutch railway network. M.Sc. thesis, Radboud University, 2016.

[24] R. P. Brent. *Algorithms for minimization without derivatives.* Dover Publications, Mineola, N.Y, 2002.

[25] Thorsten Büker and Bernhard Seybold. Stochastic modelling of delay propagation in large networks. *JRTPM*, 2(1-2):34–50, 2012.

[26] Simon Busard, Quentin Cappart, Christophe Limbrée, Charles Pecheur, and Pierre Schaus. Verification of railway interlocking systems. In Jun Pang, Yang Liu, and Sjouke Mauw, editors, *Proceedings 4th International Workshop on Engineering Safety and Security Systems, ESSS*, volume 184 of *Electronic Proceedings in Theoretical Computer Science*, pages 19–31. Open Publishing Association, 2015.

[27] Sergio Cabello, Mark de Berg, and Marc J. van Kreveld. Schematization of networks. *Comput. Geom.*, 30(3):223–228, 2005.

[28] Aaron Calafato, Christian Colombo, and Gordon J. Pace. A controlled natural language for tax fraud detection. In Davis et al. [40], pages 1–12.

[29] John J. Camilleri. GF Syntax Editor. Online: `http://cloud.grammaticalframework.org/syntax-editor/editor.html`, 2012. Accessed 2017-11-20.

[30] John J. Camilleri, Gordon J. Pace, and Michael Rosner. Controlled natural language in a game for legal assistance. In Rosner and Fuchs [148], pages 137–153.

[31] John J. Camilleri, Gabriele Paganelli, and Gerardo Schneider. A CNL for contract-oriented diagrams. In Davis et al. [39], pages 135–146.

[32] Michael Cashmore, Maria Fox, Derek Long, and Daniele Magazzeni. A compilation of the full PDDL+ language into SMT. In Amanda Jane Coles, Andrew Coles, Stefan Edelkamp, Daniele Magazzeni, and Scott Sanner, editors, *International Conference on Automated Planning and Scheduling, ICAPS 2016*, pages 79–87. AAAI Press, 2016.

[33] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2013*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.

[34] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV '00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, 2000.

[35] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19:7–34, 2001.

[36] Edmund M. Clarke and Jeanette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, December 1996. Available also as technical report CMU-CS-96-178.

[37] Arjeh Cohen, Hans Cuypers, Karin Poels, Mark Spanbroek, and Rikko Verrijzer. WExEd — WebALT exercise editor for multilingual mathematics exercises. In Mika Seppälä, Sebastian Xambo, and Olga Caprotti, editors, *WebALT Conference and Exhibition (WebALT 2006)*, pages 141–145, 2006.

[38] COIN-OR project. CBC integer programming solver v2.9.9. `https://projects.coin-or.org/Cbc`, July 2018.

[39] Brian Davis, Kaarel Kaljurand, and Tobias Kuhn, editors. *4th International Workshop ON Controlled Natural Language (CNL)*, volume 8625 of *Lecture Notes in Computer Science*. Springer, 2014.

[40] Brian Davis, Gordon J. Pace, and Adam Z. Wyner, editors. *Controlled Natural Language - 5th International Workshop, CNL 2016, Aberdeen, UK, July 25-27, 2016, Proceedings*, volume 9767 of *Lecture Notes in Computer Science*. Springer, 2016.

[41] Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors. *Datalog Reloaded. First International Workshop 2010*, volume 6702 of *Lecture Notes in Computer Science*. Springer-Verlag, 2011.

[42] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[43] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[44] Olga De Troyer, Frederik Van Broeckhoven, and Joachim Vlieghe. Creating story-based serious games using a controlled natural language domain specific modeling language. In Minhua Ma and Andreas Oikonomou, editors, *Serious Games and Edutainment Applications: Volume II*, pages 567–603. Springer International Publishing, Cham, 2017.

[45] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom.*, 4:235–282, 1994.

[46] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice-Hall, 1999.

[47] Stefan Dillmann and Reiner Hähnle. Automated planning of ETCS tracks. In *Reliability, Safety, and Security of Railway Systems, RSSRail 2019 Proceedings, to appear*, Lecture Notes in Computer Science. Springer, 2019.

[48] Véronique Donzeau-Gouge, Gilles Kahn, Bernard Lang, and Bertrand Mélèse. Document structure and modularity in mentor. In William E. Riddle and Peter B. Henderson, editors, *Software Engineering Symposium on Practical Software Development Environments*, pages 141–148. ACM, 1984.

[49] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, 1979.

[50] Joseph S Dumas and Janice Redish. *A practical guide to usability testing*. Intellect books, 1999.

[51] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th Conf., CAV 2014*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014.

[52] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT conference 2003*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[53] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.

[54] Cindy Eisner. Using symbolic model checking to verify the railway stations of Hoorn-Kersenboogerd and Heerhuowaard. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99*, volume 1703 of *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, 1999.

[55] Jason Eisner and Nathaniel W. Filardo. Dyna: Extending Datalog for modern AI. In de Moor et al. [41], pages 181–220.

[56] A. Fantechi, W. Fokkink, and A. Morzenti. Some trends in formal methods applications to railway signalling. In S. Gnesi and T. Margaria, editors, *Formal Methods for Industrial Critical Systems*, pages 61–84. John Wiley & Sons Inc., 2012.

[57] A. Ferrari, G. Magnani, D. Grasso, and A. Fantechi. Model checking interlocking control tables. In Eckehard Schnieder and Géza Tarnai, editors, *8th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2010)*, pages 107–115. Springer-Verlag, 2011.

[58] Martin Fowler. *Domain-Specific Languages.* Addison-Wesley Professional, 2010.

[59] Maria Fox and Derek Long. Modelling mixed discrete-continuous domains for planning. *J. Artif. Intell. Res.*, 27:235–297, 2006.

[60] Martin Franzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.

[61] Mitsuyoshi Fukuda, Yuji Hirao, and Takahiko Ogino. VDM specification of an interlocking system and a simulator for its validation. In U. Becker and Edward L. Schneider, editors, *9th IFAC Symposium Control in Transportation Systems*, pages 218–223. IFAC, 2000.

[62] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction*, volume 7898 of *Lecture Notes in Computer Science*, pages 208–214. Springer, 2013.

[63] Martin Gebser, Tomi Janhunen, and Jussi Rintanen. SAT modulo graphs: Acyclicity. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, volume 8761 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 2014.

[64] S. Gnesi, G. Lenzini, D. Latella, C. Abbaneo, A. Amendola, and P. Marmo. Automatic Spin validation of a safety critical railway control system. In *IEEE Conference on Dependable Systems and Networks*, pages 119–124. IEEE Computer Society Press, 2000.

[65] J. F. Groote, S. F. M. van Vlijmen, and J. W. C. Koorn. The safety guaranteeing system at station hoorn-kersenboogerd. In *COMPASS '95 – Conference on Computer Assurance Systems Integrity, Software Safety and Process Security*, pages 57–68. IEEE, June 1995.

[66] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In Peter Buneman and Sushil Jajodia, editors, *SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 157–166. ACM, 1993.

[67] Stephen Guy and Rolf Schwitter. Architecture of a web-based predictive editor for controlled natural language processing. In Davis et al. [39], pages 167–178.

[68] Catalina Hallett. Generic querying of relational databases using natural language generation techniques. In Nathalie Colineau, Cécile Paris, Stephen Wan, Robert Dale, and Anja Belz, editors, *INLG 2006 - Proceedings of the Fourth International Natural Language Generation Conference*, pages 95–102. The Association for Computer Linguistics, 2006.

[69] Stefan Hammerich. *Menübasierte Generierung natürlicher Sprache.* Project thesis (studienarbeit), University of Hamburg, 1999.

[70] I. A. Hansen and J. Pachl. *Railway Timetabling and Operations.* Eurailpress, 2014.

[71] David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic.* MIT Press, 2000.

[72] T. E. Harris and F. S. Ross. Fundamentals of a method for evaluating rail net capacities. Technical report, Santa Monica, California, 1955.

[73] John V. Harrison and Suzanne W. Dietrich. Maintenance of materialized views in a deductive database: An update propagation approach. In Kotagiri Ramamohanarao, James Harland, and Guozhu Dong, editors, *Proceedings of the Workshop on Deductive Databases held in conjunction with the Joint International Conference and Symposium on Logic Programming*, volume CITRI/TR-92-65 of *Technical Report*, pages 56–65. Department of Computer Science, University of Melbourne, 1992.

[74] Steven S. Harrod. A tutorial on fundamental model structures for railway timetable optimization. *Surveys in Operations Research and Management Science*, 17(2):85 – 96, 2012.

[75] Anne E. Haxthausen and Peter H. Østergaard. On the Use of Static Checking in the Verification of Interlocking Systems. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, pages 266–278. Springer, 2016.

[76] Anne E. Haxthausen, Jan Peleska, and Ralf Pinger. Applied bounded model checking for interlocking system designs. In Steve Counsell and Manuel Núñez, editors, *Software Engineering and Formal Methods Collocated Workshops*, volume 8368 of *Lecture Notes in Computer Science*, pages 205–220. Springer-Verlag, 2014.

[77] Michael G. Hinchey and Jonathan P. Bowen, editors. *Industrial-Strength Formal Methods*. International Series in Formal Methods. Springer-Verlag, 1999.

[78] Daniel Hürlimann. *Objektorientierte Modellierung von Infrastrukturelementen und Betriebsvorgängen im Eisenbahnwesen*. PhD thesis, ETH Zurich, 2002.

[79] Yoshinao Isobe, Faron Moller, Hoang Nga Nguyen, and Markus Roggenbach. Safety and line capacity in railways – an approach in timed csp. In John Derrick, Stefania Gnesi, Diego Latella, and Helen Treharne, editors, *Integrated Formal Methods*, pages 54–68. Springer, 2012.

[80] Phillip James, Andrew Lawrence, Markus Roggenbach, and Monika Seisenberger. Towards safety analysis of ERTMS/ETCS level 2 in real-time maude. In Cyrille Artho and Peter Csaba Ölveczky, editors, *Formal Techniques for Safety-Critical Systems - Fourth International Workshop, FTSCS 2015, Paris, France, November 6-7, 2015. Revised Selected Papers*, volume 596 of *Communications in Computer and Information Science*, pages 103–120. Springer, 2015.

[81] Phillip James and Markus Roggenbach. Encapsulating formal methods within domain specific languages: A solution for verifying railway scheme plans. *Mathematics in Computer Science*, 8(1):11–38, 2014.

[82] Kristofer Johannisson. Natural language specifications. In Beckert et al. [11], pages 317–333.

[83] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 422–430. Springer, 2016.

[84] Dejan Jovanovic and Leonardo de Moura. Solving non-linear arithmetic. *ACM Comm. Computer Algebra*, 46(3/4):104–105, 2012.

[85] Kaarel Kaljurand and Tobias Kuhn. A multilingual semantic wiki based on attempto controlled english and grammatical framework. In Philipp Cimiano, Óscar Corcho, Valentina Presutti, Laura Hollink, and Sebastian Rudolph, editors, *The Semantic Web: Semantics and Big Data, 10th International Conference, ESWC 2013, Montpellier, France, May 26-30, 2013. Proceedings*, volume 7882 of *Lecture Notes in Computer Science*, pages 427–441. Springer, 2013.

[86] Eduard Kamburjan and Reiner Hähnle. Uniform modeling of railway operations. In *Formal Techniques for Safety-Critical Systems FTSCS 2016*, volume 694 of *Communications in Computer and Information Science*, pages 55–71. Springer, 2016.

[87] Eduard Kamburjan, Reiner Hähnle, and Sebastian Schön. Formal modeling and analysis of railway operations with active objects. *Sci. Comput. Program.*, 166:167–193, 2018.

[88] Karim Kanso, Faron Moller, and Anton Setzer. Automated verification of signalling principles in railway interlocking systems. In A. Miller and M. Calder, editors, *Proceedings of the Eighth International Workshop on Automated Verification of Critical Systems (AVoCS 2008)*, volume 250 of *Electronic Notes in Theoretical Computer Science*, pages 19–31. Elsevier, 2009.

[89] Finn Kensing and Jeanette Blomberg. Participatory design: Issues and concerns. *Computer Supported Cooperative Work (CSCW)*, 7(3):167–185, 1998.

[90] Janna Khegai, Bengt Nordström, and Aarne Ranta. Multilingual syntax editing in GF. In Alexander F. Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing, 4th International Conference, CICLing 2003, Mexico City, Mexico, February 16-22, 2003, Proceedings*, volume 2588 of *Lecture Notes in Computer Science*, pages 453–464. Springer, 2003.

[91] Tobias Kuhn. AceWiki: Collaborative ontology management in controlled natural language. In Christoph Lange, Sebastian Schaffert, Hala Skaf-Molli, and Max Völkel, editors, *Proceedings of the 3rd Semantic Wiki Workshop (SemWiki 2008) at the 5th European Semantic Web Conference (ESWC 2008), Tenerife, Spain, June 2nd, 2008*, volume 360 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[92] Tobias Kuhn. How controlled english can improve semantic wikis. In Christoph Lange, Sebastian Schaffert, Hala Skaf-Molli, and Max Völkel, editors, *4th Semantic Wiki Workshop (SemWiki 2009) at the 6th European Semantic Web Conference (ESWC 2009), Hersonissos, Greece, June 1st, 2009. Proceedings.*, volume 464 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

[93] Tobias Kuhn. Codeco: A practical notation for controlled english grammars in predictive editors. In Rosner and Fuchs [148], pages 95–114.

[94] Tobias Kuhn. A survey and classification of controlled natural languages. *Computational Linguistics*, 40(1):121–170, March 2014.

[95] Tobias Kuhn and Rolf Schwitter. Writing support for controlled natural languages. In *Proceedings of the Australasian Language Technology Association (ALTA) Workshop, 2008*, 2008.

[96] Michael Kölling, Neil C. C. Brown, and Amjad Altadmri. Frame-based editing. *Journal of Visual Languages and Sentient Systems*, 3, 6 2017.

[97] Alex Landex. *Methods to estimate railway capacity and passenger delays*. PhD thesis, Technical University of Denmark (DTU), 2008.

[98] T. Lecomte, L. Burdy, and M. Leuschel. Formally checking large data sets in the railways. In F. Ishikawa and A . Romanovsky, editors, *Advances in Developing Dependable Systems in Event-B. In conjunction with ICFEM 2012*, Technical Report. Newcastle University, 2012.

[99] Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.

[100] Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artif. Intell.*, 157(1-2):115–137, 2004.

[101] Peter Ljunglöf. Dialogue management as interactive tree building. In *DiaHolmia'09, 13th Workshop on the Semantics and Pragmatics of Dialogue*, Stockholm, Sweden, 2009.

[102] Peter Ljunglöf. Editing syntax trees on the surface. In *NoDaLiDa 2011*, pages 138–145, 2011.

[103] M. Lodemann, N. Luttenberger, and E. Schulz. Semantic computing for railway infrastructure verification. In *7th International Conference on Semantic Computing (ICSC)*, pages 371–376. IEEE, 2013.

[104] LUKS: Analysis of lines and junctions. `http://www.via-con.de/development/luks`, 2018.

[105] Bjørnar Luteberget, John J. Camilleri, Christian Johansen, and Gerardo Schneider. Participatory Verification of Railway Infrastructure by Representing Regulations in RailCNL. In Alessandro Cimatti and Marjan Sirjani, editors, *International Conference on Software Engineering and Formal Methods (SEFM)*, volume 10469 of *Lecture Notes in Computer Science*, pages 87–103. Springer International Publishing, 2017.

[106] Bjørnar Luteberget, Koen Claessen, and Christian Johansen. Design-time railway capacity verification using SAT modulo discrete event simulation. In Nikolaj Bjørner and Arie Gurfinkel, editors, *FMCAD 2018*. IEEE, 2018.

[107] Bjørnar Luteberget and Christian Johansen. Efficient verification of railway infrastructure designs against standard regulations. *Formal Methods in System Design*, 52(1):1–32, Feb 2018.

[108] Bjørnar Luteberget, Christian Johansen, Claus Feyling, and Martin Steffen. Rule-based incremental verification tools applied to railway designs and regulations. In John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *21st International Symposium on Formal Methods (FM)*, volume 9995 of *Lecture Notes in Computer Science*, pages 772–778. Springer-Verlag, 2016.

[109] Bjørnar Luteberget, Christian Johansen, and Martin Steffen. Rule-based consistency checking of railway infrastructure designs. In Erika Ábrahám and Marieke Huisman, editors, *12th International Conference on integrated Formal Methods (iFM 2016)*, volume 9681 of *Lecture Notes in Computer Science*, pages 491–507. Springer, 2016.

[110] Bjørnar Luteberget, Christian Johansen, and Martin Steffen. Rule-based consistency checking of railway infrastructure designs (long version). Technical report 450, University of Oslo, Dept. of Informatics, 2016.

[111] Baohua Mao, Jianfeng Liu, Yong Ding, Haidong Liu, and Tin Kin Ho. Signalling layout for fixed-block railway lines with real-coded genetic algorithms. *Hong Kong Institution of Engineers, Transactions*, 13(1):35–40, 2006.

[112] A. Mirabadi and M. B. Yazdi. Automatic generation and verification of railway interlocking tables using FSM and NuSMV. *Transport Problems: An International Scientific Journal*, 4:103–110, 2009.

[113] Teruko Mitamura, Kathryn Baker, Eric Nyberg, and David Svoboda. Diagnostics for interactive controlled language checking. In *Controlled Language Application Workshop (CLAW 2003)*, pages 237–244, 2003.

[114] Rei Miyata, Anthony Hartley, Kyo Kageura, and Cécile Paris. Evaluating the usability of a controlled language authoring assistant. *The Prague Bulletin of Mathematical Linguistics*, 108(1):147–158, 2017.

[115] Rei Miyata, Anthony Hartley, Cécile Paris, and Kyo Kageura. Evaluating and implementing a controlled language checker. In Key-Sun Choi and Sejin Nam, editors, *Controlled Language Applications Workshop (CLAW 2016)*, pages 30–35. KAIST, 2016.

[116] Markus Montigel. *Modellierung und Gewährleistung von Abhängigkeiten in Eisenbahnsicherungsanlagen.* PhD thesis, ETH Zurich, 1994.

[117] Moisés Salvador Meza Moreno and Björn Bringert. Interactive multilingual web applications with grammatical framework. In Bengt Nordström and Aarne Ranta, editors, *International Conference in Advances in Natural Language Processing (GoTAL 2008)*, volume 5221 of *Lecture Notes in Computer Science*, pages 336–347. Springer, 2008.

[118] Boris Motik, Yavor Nenov, Robert Edgar Felix Piro, and Ian Horrocks. Incremental update of datalog materialisation: the backward/forward algorithm. In Blai Bonet and Sven Koenig, editors, *Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 1560–1568. AAAI Press, 2015.

[119] K. Nakamatsu, Y. Kiuchi, W.Y. Chen, and S.L. Chung. Intelligent railway interlocking safety verification based on annotated logic program and its simulator. In *IEEE International Conference on Networking, Sensing and Control*, volume 1, pages 694–699. IEEE, March 2004.

[120] Kazumi Nakamatsu, Yosuke Kiuchi, and Atsuyuki Suzuki. EVALPSN based railway interlocking simulator. In Mircea Gh. Negoita, Robert J. Howlett, and Lakhmi C. Jain, editors, *Knowledge-Based Intelligent Information and Engineering Systems*, volume 3214 of *Lecture Notes in Artificial Intelligence*, pages 961–967. Springer-Verlag, 2004.

[121] Andrew Nash, Daniel Huerlimann, Jörg Schütte, and Vasco Paul Krauss. RailML — a standard data interface for railroad applications. In *Computers in Railways IX*, pages 233–240. WIT Press, 2004.

[122] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. Rdfox: A highly-scalable RDF store. In Marcelo Arenas, Óscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul T. Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab, editors, *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II*, volume 9367 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2015.

[123] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

[124] Ulf Nilsson and Jan Maluszynski. *Logic, Programming, and Prolog.* John Wiley & Sons, Inc., 2nd edition, 1995.

[125] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization.* Springer, New York, NY, USA, second edition, 2006.

[126] Martin Nöllenburg and Alexander Wolff. Drawing and labeling high-quality metro maps by mixed-integer programming. *IEEE Trans. Vis. Comput. Graph.*, 17(5):626–641, 2011.

[127] Martin Nöllenburg. Automated drawing of metro maps. Technical Report 25, Universität Karlsruhe, Karlsruhe, 2005.

[128] Olufolajimi Oke and Sauleh Siddiqui. Efficient automated schematic map drawing using multiobjective mixed integer programming. *Computers & OR*, 61:1–17, 2015.

[129] OpenTrack: Simulation of railway networks. `http://www.opentrack.ch/`, 2018.

[130] Muhammet Mustafa Ozdal. *Routing Algorithms for High-Performance VLSI Packaging.* PhD thesis, 2005.

[131] J. Pachl. *Railway Operation and Control.* VTD Rail Publishing, 2015.

[132] Achilleas Papakostas and Ioannis G. Tollis. Algorithms for area-efficient orthogonal drawings. *Comput. Geom.*, 9(1-2):83–110, 1998.

[133] Achilleas Papakostas and Ioannis G. Tollis. Efficient orthogonal drawings of high degree graphs. *Algorithmica*, 26(1):100–125, 2000.

[134] O. Pavlovic and H. Ehrich. Model checking PLC software written in function block diagram. In *Third International Conference on Software Testing, Verification and Validation, ICST*, pages 439–448. IEEE, 2010.

[135] Wiktor Mateusz Piotrowski, Maria Fox, Derek Long, Daniele Magazzeni, and Fabio Mercorio. Heuristic planning for PDDL+ domains. In Subbarao Kambhampati, editor, *International Joint Conference on Artificial Intelligence, IJCAI 2016*, pages 3213–3219. IJCAI/AAAI Press, 2016.

[136] Richard Power, Donia Scott, and Roger Evans. What you see is what you meant: direct knowledge editing with natural language feedback. In *ECAI*, pages 677–681, 1998.

[137] Cristian Prisacariu and Gerardo Schneider. A Dynamic Deontic Logic for Complex Contracts. *The Journal of Logic and Algebraic Programming (JLAP)*, 81(4):458–490, 2012.

[138] Bane NOR: Model of the Norwegian rail network. `http://www.banenor.no/en/startpage1/Market1/Model-of-the-national-rail-network/`, 2016.

[139] railML. The XML interface for railway applications. `http://www.railml.org`, 2016.

[140] Aarne Ranta. Grammatical framework. *Journal of Functional Programming*, 14(2):145–189, 2004.

[141] Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars.* CSLI Publications, Stanford, 2011. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

[142] Aarne Ranta. Embedded controlled languages. In Brian Davis, Kaarel Kaljurand, and Tobias Kuhn, editors, *Controlled Natural Language: 4th International Workshop*, volume 8625 of *Lecture Notes in Computer Science*, pages 1–7. Springer-Verlag, 2014.

[143] Aarne Ranta, Krasimir Angelov, Robert Höglind, Christer Axelsson, and Leif Sandsjö. A mobile language interpreter app for prehospital/emergency care. In *Medicinteknik dagarna*, Västerås, 2017.

[144] Aarne Ranta, John Camilleri, Grégoire Détrez, Ramona Enache, and Thomas Hallgren. Grammar tool manual and best practices. Technical report, MOLTO Deliverable D2.3, MOLTO Consortium, Göteborg, 2012. `http://www.molto-project.eu/biblio/deliverable/grammar-tools-and-best-practices`.

[145] Aarne Ranta, Ramona Enache, and Grégoire Détrez. Controlled language for everyday use: The MOLTO phrasebook. In *CNL 2012*, volume 7175 of *LNCS*, pages 115–136. Springer-Verlag, 2012.

[146] Eric S. Raymond. *The Art of UNIX Programming.* Pearson Education, 2003.

[147] Stewart Robinson. *Simulation: The Practice of Model Development and Use.* John Wiley & Sons, Inc., USA, 2004.

[148] Michael Rosner and Norbert E. Fuchs, editors. *Controlled Natural Language - Second International Workshop, CNL 2010, Marettimo Island, Italy, September 13-15, 2010. Revised Papers*, volume 7175 of *Lecture Notes in Computer Science.* Springer, 2012.

[149] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[150] Diptikalyan Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In Catuscia Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP*, volume 2916 of *Lecture Notes in Computer Science*, pages 392–406. Springer, 2003.

[151] Rolf Schwitter, Anna Ljungberg, and David Hood. ECOLE — a look-ahead editor for a controlled language. In *Controlled Language Applications Workshop (CLAW 2003)*, pages 141–150, 2003.

[152] Alessandro Seganti, Pawel Kaplanski, Jesus David Nuñez Campo, Krzysztof Cieslinski, Jerzy Koziolkiewicz, and Pawel Zarzycki. Asking data in a controlled way with ask data anything NQL. In Davis et al. [40], pages 58–68.

[153] Samanah Seyedi-Shandiz. Schematic representation of the geographical railway network used by the Swedish transport administration. M.Sc. thesis, Lund University, 2014.

[154] Helen Sharp, Yvonne Rogers, and Jenny Preece. *Interaction design: beyond human-computer interaction*. John Wiley, 2007.

[155] Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, 2005.

[156] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding context-sensitivity (the making of a precise and scalable pointer analysis). In Thomas Ball and Mooly Sagiv, editors, *38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL)*, pages 17–30. ACM, January 2011.

[157] Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In de Moor et al. [41], pages 245–251.

[158] Gunnar Stalmårck. A system for determining logic theorems by applying values and rules to triplets that are generated from a formula. Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 897 (approved 1994), European Patent No. 0403 454 (approved 1995), 1992.

[159] Terrance Swift. Incremental tabling in support of knowledge representation and reasoning. *Theory and Practice of Logic Programming*, 14(4-5):553–567, 2014.

[160] Terrance Swift and David S. Warren. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 12(1-2):157–187, January 2012.

[161] Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.

[162] Tim Teitelbaum and Thomas W. Reps. The cornell program synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[163] Harry R. Tennant, Kenneth M. Ross, Richard M. Saenz, Craig W. Thompson, and James R. Miller. Menu-based natural language understanding. In Mitchell P. Marcus, editor, *Annual Meeting of the Association for Computational Linguistics (ACL 1983)*, pages 151–158. ACL, 1983.

[164] C. W. Thompson. *Using Menu-based Natural Language Understanding to Avoid Problems Associated with Traditional Interfaces to Databases*. Ph.D. Dissertation, Deptartment of Computer Science, University of Texas, Austin, 1989.

[165] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems (Volume I & II)*. Computer Society Press, 1988.

[166] Frederik Van Broeckhoven, Joachim Vlieghe, and Olga De Troyer. Using a controlled natural language for specifying the narratives of serious games. In Henrik Schoenau-Fog, Luis Emilio Bruni, Sandy Louchart, and Sarune Baceviciute, editors, *International Conference on Interactive Digital Storytelling (ICIDS 2015)*, volume 9445 of *Lecture Notes in Computer Science*, pages 142–153. Springer, 2015.

[167] Cristina Vertan and Walther von Hahn. Menu choice translation — a flexible menu-based controlled natural language system. In *Controlled Language Application Workshop (CLAW 2003)*, 2003.

[168] Linh Hong Vu, Anne Elisabeth Haxthausen, and Jan Peleska. A domain-specific language for railway interlocking systems. In *FORMS/FORMAT 2014*, pages 200–209. TU Braunschweig, 2014.

[169] Martin P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.

[170] Jim Welsh, Brad Broom, and Derek Kiong. A design rationale for a language-based editor. *Software: Practice and Experience*, 21(9):923–948, 1991.

[171] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog with binary decision diagrams for program analysis. In K. Yi, editor, *Third Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–108. Springer-Verlag, 2005.

[172] K. Winter, W. Johnston, P. Robinson, P. Strooper, and L. van den Berg. Tool support for checking railway interlocking designs. In Tony Cant, editor, *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software*, pages 101–107. ACM, 2006.

[173] Kirsten Winter. Optimising ordering strategies for symbolic model checking interlocking control tables. In T. Margaria and B. Steffen, editors, *5th International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation (ISOLA'12), Part II*, volume 7610 of *Lecture Notes in Computer Science*, pages 246–260. Springer-Verlag, 2012.

[174] Alexander Wolff. Drawing subway maps: A survey. *Inform., Forsch. Entwickl.*, 22(1):23–44, 2007.

[175] Adam Z. Wyner, Krasimir Angelov, Guntis Barzdins, Danica Damljanovic, Brian Davis, Norbert E. Fuchs, Stefan Höfler, Ken Jones, Kaarel Kaljurand, and Tobias Kuhn. On controlled natural languages: Properties and prospects. In *CNL 2009*, volume 5972 of *LNCS*, pages 281–289. Springer, 2009.

[176] Peter J. Zwaneveld, Leo G. Kroon, and Stan P.M. van Hoesel. Routing trains through a railway station based on a node packing model. *European Journal of Operational Research*, 128(1):14 – 33, 2001.